VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Rozpoznávání tváří pomocí konvolučních neuronových sítí

Face Recognition Using Convolutional Neural Network

2020 Jan Kněžík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Kněžík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Rozpoznávání tváří pomocí konvolučních neuronových sítí
Face Recognition Using Convolutional Neural Network

Jazyk vypracování: čeština

Zásady pro vypracování:

Rozpoznání obličejů v obrazech je v posledních letech hodně rozvíjené téma. Aplikace tohoto druhu může být použita například v oblasti bezpečnosti.

1. Popište základní pojmy a metody v oblasti rozpoznávání objektů v obrazech. Zaměřte se zejména na lidské tváře.
2. Seznamte se s volně dostupnými knihovnami a popište jaké možnosti nabízí v této oblasti (OpenCV, Dlib, TensorFlow).
3. S pomocí knihoven vytvořte vybraný detektor (rozpoznávač) tváří.
4. Experimentálně ověřte funkčnost, přesnost a rychlost navrženého řešení na renomovaných datasetech.
5. Své závěry řádně zdokumentujte v textu práce.

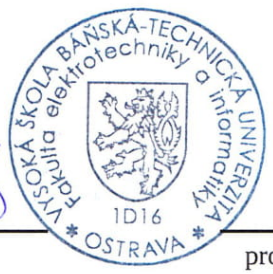
Seznam doporučené odborné literatury:

[1] C. Szegedy et al., "Going deeper with convolutions," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 1-9.
[2] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 770-778.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Radovan Fusek, Ph.D.**

Datum zadání: 01.09.2019
Datum odevzdání: 30.04.2020



doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 15. května 2020 .....

Rád bych poděkoval vedoucímu diplomové práce Ing. Radovanu Fuskovi, Ph.D. za odborné vedení a cenné rady při zpracování této práce.

Abstrakt

Tato diplomová práce se zabývá rozpoznáváním obličejů pomocí konvolučních neuronových sítí. V rámci teoretické části jsou popsány konvoluční sítě společně s metodami a architekturami, které umožnují natrénovat hluboké a kvalitní modely neuronových sítí. V praktické části jsou využity všechny metody a architektury z teoretické části pro natrénování mnoha modelů a ty jsou poté porovnány na testovacím datasetu Labeled Faces in the Wild pro verifikaci obličejů.

Klíčová slova: rozpoznání obličejů, konvoluční neuronové sítě, klasifikace, strojové učení, počítačové vidění, Keras

Abstract

This master thesis deals with face recognition using convolutional neural networks. In the theoretical part, convolutional networks are described together with methods and architectures, which allow training of deep and powerful neural network models. All these methods and architectures are used for training many models, which are then tested and compared on benchmark dataset for face verification called Labeled Faces in the Wild.

Key Words: face recognition, convolutional neural network, classification, machine learning, computer vision, Keras

Obsah

Seznam použitých zkratek a symbolů

KNS	–	Konvoluční neuronová síť
SGD	–	Stochastic gradient descent
ILSVRC	–	ImageNet Large Scale Visual Recognition Challange
ReLU	–	Rectified linear unit
LFW	–	Labeled Face in the Wild
LBP	–	Local binary patterns
SIFT	–	Scale-invariant feature transform
HOG	–	Histogram of Oriented Gradients
GPU	–	Graphics processing unit
MTCNN	–	Multi-task Cascaded Convolutional Network
NMS	–	Non-maximum suppression
LRN	–	Local Response Normalization

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Rozpoznání člověka pomocí obličeje je jedna z mnoha biometrických metod. První spolehlivou metodou, jak identifikovat jedince, byl otisk prstu. Avšak jedna z mnoha výhod u rozpoznání pomocí obličeje je, že není potřebná účast daného jedince. Z tohoto důvodu je poptávka po systémech schopné rozpoznávat tváře v sektoru bezpečnosti, například na letištích. Zatím je však mnohem spolehlivější rozpoznávání tváří dobrovolných účastníků. Takové aplikace slouží například pro verifikaci uživatelů, místo potřeby pro pamatování si hesla. Rozpoznávání obličejů lze rozdělit do dvou kategorii - identifikace a verifikace obličejů. U identifikace je obličeji na obrázku přiřazená identita z databáze známých identit. Zato verifikace obličejů je binární klasifikace, u které se potvrzuje, nebo zamítá, jestli obličej na obrázku patří jisté osobě.

Jako u mnoha oblastí, kde se řeší problém klasifikace dat, tak i u rozpoznání obličejů se již skoro výhradně používá hluboké učení (deep learning). Pod hluboké učení spadají metody založené na umělých neuronových sítích, které se skládají z mnoha vrstev. Jedna z těchto metod jsou konvoluční neuronové sítě, které jsou náplní této práce. I když byla konvoluční síť poprvé komerčně použita již v 90. letech [1], tak k jejich vzrůstu v popularitě nedošlo až do roku 2012. K této popularitě přispěl hlavně nárůst výkonů počítačů a paralelizace výpočtu na GPU. V porovnání s tradičními přístupy v oblasti strojového učení, dosahuje hluboké učení daleko lepší přesnosti. Na druhou stranu je ale pro natrénování neuronových sítí potřeba velké množství dat a proto taky hrají velkou roli v pokroku sítí velké veřejné datasety s označenými daty.

Dalo by se říct, že tato práce navazuje na moji bakalářskou práci, kde jsem popsal a experimentoval se staršími tradičními metodami pro extrakci příznaku a klasifikaci obličejů. Také jsem lehce popsal konvoluční neuronové sítě, ale v rámci praktické části nebyly použity. Tato práce je konkrétně zaměřena na konvoluční neuronové sítě, které jsou v posledních letech standardem v oblasti rozpoznávání objektů v obrazech.

V kapitole 2 je lehce nastíněná historie a také způsob, jak se v dnešní době používají konvoluční neuronové sítě v oblasti rozpoznávání obličejů.

V kapitole 3 jsou podrobně popsány základní vrstvy v konvolučních neuronových sítích.

Kapitola 4 se týká trénování neuronových sítí a hlavně jsou rozebrané chybové funkce, které se používají u konvolučních sítí v oblasti rozpoznávání obličejů.

Kapitola 5 je zaměřena na regularizační techniky, které slouží pro předcházení přetrénování, což je jeden z hlavních problémů při trénování neuronových sítí.

V kapitole 6 jsou popsány další běžné praktiky, které se aplikují za účelem lepšího průběhu trénování a kvalitnějšího finálního modelu neuronové sítě.

Kapitola 7 pojednává o architekturách konvolučních sítí, na které byla zaměřená velká pozornost v průběhu vývoje oblasti rozpoznávání objektů v obrazech.

Kapitola 8 je věnovaná praktické části této práce. To zahrnuje popis implementace, použitých knihoven a datasetů a také jak probíhalo předzpracování dat, trénování a testování konvolučních sítí.

2 Úvod do problematiky rozpoznávání obličejů

Obvyklý postup pro rozpoznání obličeje se skládá ze čtyř fází: detekce obličeje, zarovnání obličeje, extrakce příznaků a klasifikace. Snad nejdůležitější fází je extrakce příznaků, ve které se musí obstarat dostatečně separabilní příznaky, aby podle nich šlo tváře rozdělit do jednotlivých tříd. Proto je extrakce příznaků pro rozpoznání obličejů dlouho zkoumané odvětví počítačového vidění, které se výrazně posunulo kupředu po představení metody Eigenface [3] v 90. letech. Tato metoda a podobné holistické přístupy [4] byly v počátcích 21. století nahrazeny či skombinovány s ručně vytvořenými a lokálně založenými metodami, jako je například LBP [5], SIFT [6] nebo HOG [7]. Avšak na rozdíl od tradičních ručně vytvořených příznaků, mnohem sofistikovanější příznaky mohou být naučeny přímo z trénovacích dat.

Po velkém úspěchu konvoluční neuronové sítě AlexNet [46] v soutěži ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 [48] v kategorii klasifikace obrazů se začala oblast počítačového vidění díky hlubokému učení vyvíjet nesmírným tempem. Tento vývoj také neminul metody pro rozpoznání obličejů a v roce 2014 vyšel systém založený na konvolučních neuronových sítích pod názvem DeepFace [9] od výzkumného týmu Facebook AI. Tento systém byl snad první velký průlom v rozpoznání obličejů s použitím hlubokého učení a dosáhl téměř stejného výkonu (97.35%) jako člověk (97.53%) na standardním benchmark datasetu Labeled Faces in the Wild (LFW) [10] pro verifikaci obličejů v neomezených (reálných) podmínkách. Netrvalo dlouho a lidský výkon na LFW byl překonán systémy jako jsou DeepID2 (99.15%) [12] a FaceNet (99.63%) [13].

U klasické klasifikaci obrazů pomocí konvolučních neuronových sítí funguje síť jako algoritmus, který zahrnuje extrakci příznaku i klasifikaci. Výstupem sítě je tedy již označení třídy, ale tato třída musí být zastoupena v průběhu trénování sítě. To však není u praktických aplikací pro rozpoznávání obličejů možné a většinou je nutné provést identifikaci nebo verifikaci na identitách, které síť ještě neviděla. Označení třídy, které reprezentuje výstup sítě tedy nenese užitečnou informaci. Síť ale stále extrahuje užitečné příznaky, takže se při testování odstraňuje poslední vrstva sítě a výstup předposlední vrstvy se bere jako reprezentace tváře neboli vektor příznaků. Z toho důvodu je vývoj hlubokého učení v oblasti rozpoznávání tváří zaměřený na navržení chybových funkcí, které pomohou vytvořit více diskriminační příznaky (viz podkapitola 4.1), zatímco vývoj v obecnější oblasti klasifikace obrazů je z velké části zaměřený na navržení lepších architektur (viz kapitola 7). Konvoluční neuronové sítě se u rozpoznání tváří dají vnímat jako deskriptory příznaků a kvalita těchto příznaků záleží na velikosti a kvalitě trénovacího datasetu.

Všechny výše zmíněné systémy (DeepFace, DeepID2, FaceNet) však byly natrénovány na obrovských soukromých datasetech, tudíž jejich výsledky nebylo možné reprodukovat. Ruku v ruce s vývojem rozpoznání obličejů v hlubokém učení šel tedy i vývoj veřejných datasetů (viz kapitola 8.3).

3 Konvoluční neuronové sítě

V této kapitole jsou popsány jednotlivé vrstvy v konvoluční neuronové síti. Celá architektura sítě je pak různě složena z těchto vrstev, které se mohou opakovat a jejich kombinace se liší podle navržení architektury.

3.1 Konvoluční vrstva

Na rozdíl od vrstev v regulérních umělých neuronových sítích má konvoluční vrstva uspořádané neurony ve 3 dimenzích: výška, šířka a hloubka. Neurony společně v jedné 2D matici tvoří **příznakovou mapu**. Jelikož vstupem konvolučních sítí jsou vysoce dimenzionální data jako obrázky, tak není praktické propojit všechny neurony mezi sousedními vrstvami. Proto je příznaková mapa vytvořena pomocí **konvolučního filtru**, s kterým se provádí diskrétní konvoluce mezi vstupní maticí a daným filtrem (rovnice 1). Hodnoty v konvolučním filtru jsou trénovatelné váhy, které jsou tímto způsobem sdílené pro všechny neurony ve stejné příznakové mapě. Podle stanoveného počtu filtrů je vytvořený stejný počet příznakových map.

$$(f * g)_{x,y} = \sum_i \sum_j f_{x-i,y-j} \cdot g_{i,j} \tag{1}$$

Jelikož se filtr posouvá po celém vstupu, tak musí jeho hloubka být stejná jako počet kanálu vstupu, tzn. když je vstup $m \times n \times k$ a velikost filtrů je nastavený pro danou vrstvu na $a \times b$, tak při použití musí být filtry velké $a \times b \times k$.

Po konvoluci se dále k hodnotě neuronu přičte **bias**, který dokáže posouvat aktivační funkci po ose x. Bias může být buďto vázaný, nebo nevázaný. Vázaný bias je stejný pro celou příznakovou mapu a u nevázaného biasu má každý neuron v příznakové mapě vlastní bias.

Předtím než jsou hodnoty z příznakových map poslány do další vrstvy, tak vstupují do **aktivační (přenosové) funkce**. Aktivační funkce slouží pro zajištění nelineárnosti sítě, což je velice důležitá vlastnost, jelikož reálná data nejsou lineárně separabilní. Tyto funkce se aplikují u každého neuronu, nejenom u konvolučních, ale také i u plně propojených vrstev. Umělé neuronové sítě byly inspirovány z biologických neuronových sítí, ve kterých se domnívalo že neuron se "aktivuje" pokud je jeho vstupní signál dostatečně velký. To by se dalo reprezentovat jako výstup neuronu 0 pokud je neaktivní a 1 pokud je aktivní. Přesně z toho důvodu se v prvních umělých neuronových sítích používala binární skoková aktivační funkce. Avšak poté co se u trénování sítí začalo používat zpětné šíření, tak skokovou funkci již nebylo možné používat, protože není diferencovatelná. Proto začala být používána její aproximace – funkce sigmoid: $f(x) = \frac{1}{1+e^{-x}}$. Později se začala používat funkce tanh (hyperbolická tangens): $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ s oborem hodnot <-1, 1> oproti funkci sigmoid s oborem hodnot <0, 1>. Tohle vycentrování kolem 0 vede k rychlejší konvergenci [28]. V dnešní době je skoro výhradně používaná funkce **Rectified Linear Unit (ReLU)**: $f(x) = max(0, x)$ a nebo nějaká obdobná varianta jako například Parametric ReLU (PReLU). Síť s ReLU funkcemi se v porovnání se saturačními funkcemi

trénuje rychleji [46] a také předchází problému mizícího gradientu. Nevýhodou je však výskyt problému zvaný Dying ReLU problem. Více o těchto problémech a jak ovlivňují trénování je uvedeno v podkapitole 4.4.

Výstup *j-té* příznakové mapy v konvoluční vrstvě tedy vypadá následovně:

$$X_j^l = f(\sum_{i=1}^{N} X_i^{l-1} * W_{ij}^l + b_j^l), \tag{2}$$

kde l označuje vrstvu, W jsou konvoluční filtry (matice vah), $*$ je 2D konvoluce (rovnice 1), N je počet kanálu/příznakových map vrstvy $l-1$, b je vázaný bias a f je aktivační funkce.

Každá konvoluční vrstva sdílí společnou charakteristiku, která se stává více komplexnější a méně lokálně prostorová hlouběji v síti. První vrstvy mohou detekovat nízko úrovňové příznaky jako například hrany, zatímco hlubší vrstvy zachycují komplikované vzory jako například oči a nos. Na obrázku 1 jde vidět vizualizace vrstev naučené sítě, kde pro každou vrstvu jsou náhodně vybrány 4 příznakové mapy a pro každou příznakovou mapu jsou zobrazeny "recepční pole" 9 neuronů s největšími aktivačními hodnotami. Tyto "recepční pole" jsou doopravdy projekce aktivačních hodnot neuronů do pixelového prostoru pomocí dekonvoluční sítě. Více o této vizualizaci lze najít v práci [2].

Obrázek 1: Vizualizace příznaků v naučené síti z práce [2]

Hyperparametry této vrstvy jsou počet aplikovaných filtrů, velikost filtrů, zero-padding a krok. Podle počtu filtrů se vytvoří stejný počet příznakových map, krok určí po kolika pixelech se bude filtr při konvoluci posouvat a zero-padding doplní vstupní matici kolem krajů nulami, což umožní zachovat výšku a šířku vstupní matice (při kroku 1). Na základě těchto hyperparametrů lze vypočítat velikost příznakové mapy:

$$O = \frac{W - K + 2P}{S} + 1, \tag{3}$$

kde O je výška/šířka příznakové mapy, W je výška/šířka předchozí vrstvy, K je výška/šířka

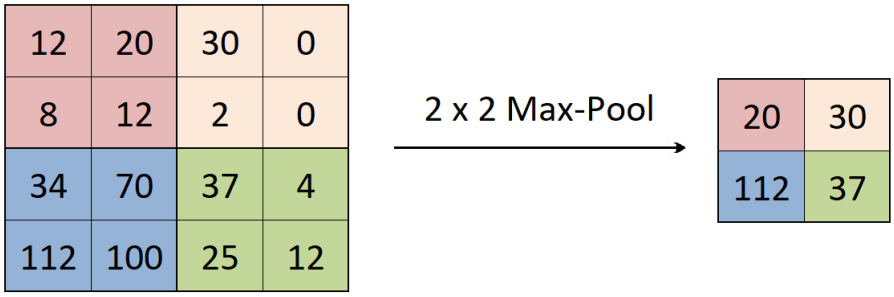
filtru, P je velikost okraje vyplněný nulami(zero-padding) a S je velikost kroku. Jakmile je znám velikost příznakové mapy, tak lze také vypočítat počet neuronů ve vrstvě:

$$N = F \cdot Ov \cdot Oš, \tag{4}$$

kde F je počet filtrů, Ov je výška a Oš je šířka příznakové mapy.

3.2 Pooling vrstva

Pooling vrstva, většinou periodicky aplikovaná po několika konvolučních vrstvách, slouží pro redukci prostorové velikosti zmenšením příznakové mapy. Tím se zmenší výpočetní náročnost a také přidá síti invarianci vůči posunu objektu ve vstupu. Podobně jako u konvolučních vrstev by se zde dalo představit posuvné okno, které se posouvá po vstupu určitým krokem a výstupem každé podoblasti je jedna hodnota. Okno se posouvá po každé vstupní matici zvlášť, takže je zachována stejná hloubka vrstvy. Na rozdíl od konvoluční vrstvy se v této vrstvě provádí vždycky stejná operace, takže vrstva nemá žádné trénovatelné parametry. Taková operace je buďto **average pooling**, která vypočítá průměrnou hodnotu a nebo **max pooling** (obrázek 2), který vybere pouze tu největší hodnotu. Lepších výsledků ale většinou dosahuje max pooling. Podoblasti se mohou překrývat a to záleží na volbě velikosti kroku a velikosti posuvného okna. Většinou se používají malá okna, abychom nepřišli o důležité informace, např. 3×3 s krokem 2.



Obrázek 2: Výsledek 2×2 maxpool vrstvy s krokem 2

3.3 Plně propojená vrstva

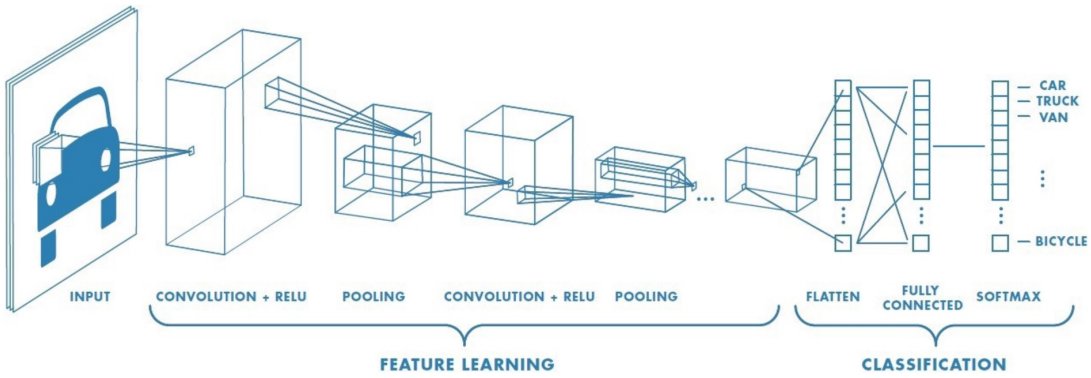
Na konci sítě se většinou používají plně propojené (fully connected) vrstvy, u kterých je každý neuron napojený na každý neuron předchozí vrstvy. Celá vrstva je tak reprezentována jako vektor a pokud je předešlá vrstva konvoluční nebo pooling, tak je nutné upravit vícerozměrná data do jednorozměrného vektoru. Někdy se tato operace zploštění vyskytuje jako vrstva sama o sobě, která je nazvaná *Flatten*. Stejně jako u konvolučních vrstev se u plně propojených vrstev používá bias a nelineární aktivační funkce.

Poslední plně propojená vrstva je **výstupní vrstva**, která má při vícetřídní klasifikaci stejný počet neuronů jako je počet tříd a při binární klasifikaci má jeden neuron. Při klasické vícetřídní klasifikaci se v této vrstvě většinou používá aktivační funkce **Softmax**:

$$f(x) = \frac{e^{x_t}}{\sum_{j=1}^{C} e^{x_j}}, \tag{5}$$

kde x je vstup do aktivační funkce Softmax, C je počet tříd a t je index skutečné třídy. Výstup poslední vrstvy se Softmax funkcí je N dimenzionální vektor, kde N je počet tříd a každá hodnota vektoru reprezentuje s jakou pravděpodobností patří obraz do dané třídy. Hodnoty jsou v rozmezí <0, 1> a součet všech hodnot činí 1. Při binární klasifikaci se jako aktivační funkce používá sigmoid, která je ekvivalentní s funkcí Softmax při dvou třídách. Menší výhodou funkce sigmoid je potřeba pouze jednoho výstupního neuronu místo dvou a tím pádem je potřeba menší počet parametrů. U rozpoznání obličejů je však konec sítě o něco složitější a tato problematika je podrobněji popsána v podkapitole 4.1.

Jednoduchá konvoluční neuronová síť, složená ze všech výše zmíněných vrstev, jde vidět na obrázku 3.



Obrázek 3: Jednoduchá konvoluční neuronová síť [53]

4 Trénování konvoluční neuronové sítě

Konvoluční neuronové sítě se převážně používají pro klasifikaci objektů v obrazech. Obecně je úkolem klasifikace předpovědět označení třídy vstupního objektu. Tento druh problému spadá pod metodu učení s učitelem, kde každý vzorek z trénovacích dat je tvořen vstupním objektem a požadovaným výstupem klasifikátoru. Po natrénování klasifikátoru na trénovacích datech je v optimálním případě klasifikátor schopný generalizovat, tzn. správně klasifikovat dosud nespatřená data. Cílem této kapitoly je popsat, jak takové trénování probíhá u neuronových sítí. Hlavní důraz je kladen na chybové funkce, které hrají velkou roli v oblasti rozpoznávání obličejů pomocí KNS. Dále je popsaná optimalizace chybové funkce pomocí algoritmu Stochastic gradient descent a jaké problémy při trénování můžou způsobit aktivační funkce.

4.1 Chybová funkce

Chybová funkce vyjadřuje míru chyby sítě. Vypočítáním chybové funkce je výsledkem chyba pro jeden trénovací vzor a globální chyba sítě je vypočítána aritmetickým průměrem chyb z celé trénovací sady. Trénování sítě probíhá minimalizací globální chyby, což jde v době trénování provést pouze změnou trénovatelných parametrů. Jinak řečeno, úkolem optimalizačních algoritmů je najít (nejlépe globální) minimum chybové funkce, kde prostor funkce tvoří parametry sítě.

Volba chybové funkce záleží na řešeném problému. Při klasické klasifikaci objektů se převážně používá **křížová entropie**: $H(p,q) = -\sum_i p_i \log(q_i)$, kde odhadnuté rozdělení pravděpodobnosti q_i reprezentuje výstup sítě a skutečné rozdělení pravděpodobnosti p_i reprezentuje skutečné označení objektu. Při kombinaci aktivační funkce Softmax (rovnice 5) a křížové entropie se chybové funkci také říká **Softmax loss**:

$$L = -\log \frac{e^{f_t}}{\sum_{j=1}^{C} e^{f_j}}. \tag{6}$$

Odhadnuté rozdělení pravděpodobnosti q_i neboli výstup poslední vrstvy s aktivační funkcí Softmax má stejnou velikost jako je počet tříd, takže skutečné označení objektů musí být také ve formě vektoru o velikosti počtu tříd. Proto se označení převádí do *kódu 1 z N* a výsledek kódování je vektor o velikosti N s hodnotami 0 na všech pozicích, kromě jedné pozice na které je hodnota 1 a tato pozice je unikátní pro danou třídu.

Při klasické klasifikaci objektů se třídy testovacích dat nachází v trénovací sadě, což značí "closed-set" testovací protokol. U rozpoznání tváří se testovací protokol od klasické klasifikace objektů liší, jelikož při praktické aplikaci systému nejsou většinou identity testovaných obličejů v trénovací sadě. Proto se při testování sítě na benchmark datasetech využívá "open-set" testovací protokol, kde žádná identita v trénovacím datasetu se nesmí nacházet v testovacím datasetu. Taková generalizace i na jiné třídy je proveditelná, jelikož lidské obličeje sdílí podobný tvar, texturu a tak dále. Tento způsob testování ale znemožní využití predikce identity z výstupu

sítě. Proto se při testování odstraňuje poslední vrstva sítě a výstup předposlední vrstvy tvoří reprezentaci tváře neboli vektor příznaků.

I když jsou naučené příznaky pomocí Softmax loss dostatečné separabilní pro identity v trénovacím datasetu, tak ale nejsou dostatečně diskriminační na rozpoznání obličejů neznámých identit v testovacím datasetu. Diskriminační příznaky jsou charakterizované velkým mezi-třídním rozptylem a také co nejmenším rozptylem v jednotlivých třídách. Tohle je hlavní problém při rozpoznání obličejů v reálných podmínkách, protože rozdíly v jednotlivých třídách můžou být větší než mezi-třídní rozdíly. Diskriminační příznaky mohou být při identifikaci obličejů klasifikovány algoritmem k-nejbližších sousedů, u kterého není potřeba trénování modelu a tím pádem nejsou předem stanovené třídy. V případě verifikace je potřeba určit práh a na jeho základě se provádí binární klasifikace, jestli obličeje patří stejné osobě nebo ne. Jelikož pro vytváření takových diskriminačních příznaků nebyla funkce Softmax loss navržena, tak se začali vymýšlet nové chybové funkce, přímo pro natrénování sítí pro rozpoznání obličejů.

Chybové funkce, které začaly být používány na počátku vývoje hlubokého učení v oblasti rozpoznání obličejů, by se daly zařadit pod *metric learning*, kde se síť neučí predikovat třídy, ale přímo se učí najít diskriminační reprezentaci dat pomocí nějaké metriky. Přesněji je metric learning u rozpoznání obličejů proces, ve kterém se minimalizuje diskriminační chybová funkce, která vede ke zmenšení dané metriky pro páry obličejů stejné osoby a zvětšení pro páry různých osob. První taková široce používaná funkce je **contrastive loss** [11]:

$$L = y_{ij} \cdot d(f(x_i), f(x_j)) + (1 - y) \cdot max(0, m - d(f(x_i), f(x_j))), \tag{7}$$

kde $f(x_i)$ a $f(x_j)$ jsou vektory příznaků dvou obrazů obličejů, d je metrika, m je hodnota označující jak daleko by od sebe měli být obličeje odlišných osob a $y_{ij} = 1$ pokud x_i a x_j patří stejné osobě a $y_{ij} = 0$ pokud nepatří. Cílem funkce je tedy najít reprezentaci s co nejmenší vzdáleností d pro pozitivní páry a větší vzdálenost než m pro negativní páry. Například v systému DeepID2 [12] byla tato funkce skombinovaná společně se Softmax loss. Další chybová funkce **triplet loss** byla navržena současně se systémem FaceNet [13] a vyjadřuje chybu na základě trojice obličejů. Funkci lze definovat následovně:

$$L = max(0, m + d(f(x_a), f(x_p)) - d(f(x_a), f(x_n))), \tag{8}$$

kde obličej na obrazu x_a je "anchor", obličej na obrazu x_p patří stejné osobě jako "anchor" a obličej na obrazu x_n patří jiné osobě. Cílem funkce je najít takovou reprezentaci obličeje, která má o m menší vzdálenost k x_a, než k x_n. Na rozdíl od contrastive loss, která bere v úvahu absolutní vzdálenost pozitivních a negativních párů, tak triplet loss pracuje s jejich relativními vzdálenostmi. Tato funkce může být použitá při optimalizaci samotně a nebo lze také prvně natrénovat síť se Softmax loss a poté ji doladit s triplet loss [14]. Nevýhoda funkcí triplet a contrastive loss je obtížné zvolení správných párů/trojic obrazů obličejů a kvalita natrénované sítě je na tomto výběru velice závislá.

Chybová funkce **Center loss** byla navržená v práci [15] a její definice je následovná:

$$L = \frac{1}{2}\sum_{i=1}^{n} ||f_i - c_{y_i}||_2^2, \tag{9}$$

kde n je velikost vektoru příznaků f a c_y je centr třídy y. Tato funkce penalizuje vzdálenost mezi příznaky a jejich odpovídajícími třídními centry v Euklidovském prostoru. Jak se tyto centra počítají během trénování sítě, je vysvětleno v kapitole 8.2, kde je zároveň ukázaná implementace Center loss. Funkce sama o sobě zmenšuje pouze rozptyl v jednotlivých třídách a proto funkci doprovází Softmax loss pro zvětšení rozptylu mezi třídami. Jelikož jsou zde použity centra tříd, které se vypočítávají během trénování sítě, tak není potřeba vytvářet žádné páry nebo trojice obrazů obličejů.

Nejnovější a efektivní chybové funkce jsou úhlově založené upravené Softmax loss funkce, které vytvářejí diskriminační vektory příznaků na nadkouli, kdežto dosavadní chybové funkce vytvářely diskriminační vektory příznaků v Euklidovském prostoru. Více dopodrobna je princip těchto funkcí vysvětlen v kapitole 8.2, kde je zároveň ukázaná implementace **Additive Angular Margin loss (ArcFace)** [16]. ArcFace je jedna z nejnovějších funkcí tohoto typu a je to vylepšení funkce Angular softmax (A-softmax) [17]. ArcFace lze vyjádřit

$$L = -\log \frac{e^{S(\cos(\theta_t + m))}}{e^{S(\cos(\theta_t + m))} + \sum_{j=1, j\neq t}^{C} e^{S\cos\theta_j}}, \tag{10}$$

kde θ_j je úhel mezi L2 normalizovanými příznaky a váhami poslední vrstvy *j-tého* neuronu, penalizační hodnota m zvětšuje úhel mezi váhami a příznaky stejné třídy a hodnota S slouží pro zvětšení hodnot.

4.2 Stochastic gradient descent

Při vytvoření sítě jsou váhy inicializovány náhodně (viz podkapitola 6.2), takže chyba sítě bude s největší pravděpodobností velká. Proto je nutné optimalizovat chybovou funkci a v neuronových sítích jsou rozhodně nejvíce používané iterační metody založené na gradientním sestupu. Základní takovou metodou je **Stochastic gradient descent (SGD)**, která byla postupem času rozšiřována (viz podkapitola 4.3) a nebo také vznikaly její nové varianty, kde nejčastěji používanou je Adaptive Moment Estimation (Adam) [21].

U SGD se chybová funkce minimalizuje s pomocí gradientu, který ukazuje v jakém směru má funkce v daném bodě největší růst. Pro vypočítání gradientů se používá **zpětné šíření (backpropagation)**, kde se postupně počítají parciální derivace chyby sítě, vzhledem ke všem váhám. Tento výpočet se provádí pomocí řetízkového pravidla (chain rule):

$$\frac{\partial L}{\partial w_{ij}^l} = \frac{\partial L}{\partial out_j^l}\frac{\partial out_j^l}{\partial net_j^l}\frac{\partial net_j^l}{\partial w_{ij}^l}, \tag{11}$$

kde w je váha, *out* je výstup neuronu, *net* je hodnota neuronu před aktivační funkcí, i je neuron vrstvy *l-1* a j je neuron vrstvy l.

Jelikož gradient ukazuje směr největšího růstu, tak pro minimalizaci chyby sítě se váhy aktualizují s použitím záporného gradientu (rovnice 12). Důležitým nastavením SGD je **learning rate** označován η, který ovlivňuje jak moc se budou měnit hodnoty vah. Příliš malá hodnota learning rate může vést k dlouhému trénování, které se může zaseknout v lokálním minimu a příliš velká hodnota vede k nestabilnímu trénování.

$$w_{ij}^{l}(t+1) = w_{ij}^{l}(t) - \eta \frac{\partial L}{\partial w_{ij}^{l}(t)} \tag{12}$$

Dále je u SGD možné nastavit **počet vzorků (batch size)** použitých v každé iteraci algoritmu. Jedna iterace znamená provedení dopředného průchodu na nastaveném počtu vzorků a jednu aktualizaci vah pomocí zpětného šíření. Po zpracování všech trénovacích vzorů uběhne jedna epocha. Podle zvoleného počtu vzorků lze algoritmus nazvat třemi různými způsoby:

- **Stochastic gradient descent** používá 1 náhodný vzorek z trénovací sady v každé iteraci
- **Batch gradient descent** používá celou trénovací sadu v každé iteraci
- **Mini-batch gradient descent** používá n náhodně vybraných vzorků v každé iteraci, kde *n>1* a *n<počet vzorků* v trénovací sadě

Použití celé trénovací sady v každé iteraci zní rozumně, jelikož je cílem minimalizovat globální chybu a (záporný) gradient vypočítaný na celé sadě směřuje vždy dolů v globální chybové funkci. Problémem v tomto případě, je ale možnost zasekávání v lokálním minimu blízko od počáteční inicializace. Naopak při počítání gradientu pro každý vzorek bude globální chyba hodně kolísat, což si lze představit jako kličkování po povrchu globální chybové funkce a díky tomu se algoritmus může dostat z lokálního minima. To se děje, jelikož (záporný) gradient sice směřuje dolů v chybové funkci pro daný vzorek, ale v globální chybové funkci může změna parametrů v daném směru znamenat větší globální chybu. U velkého počtu dat je také při počítání gradientu pro každý vzorek výhodou rychlejší konvergence, kvůli mnohokrát častější aktualizaci parametrů. K pravé konvergenci ale nikdy nedojde, jelikož algoritmus bude pořád kličkovat v oblasti (optimálně globálního) minima. Kompromis mezi těmito volbami je *mini-batch gradient descent*, kde gradient z celého mini-batch je jakási aproximace gradientu celé sady a zároveň v něm zůstává určitý potřebný "šum". Hlavní výhodou oproti SGD s 1 vzorkem je však možnost zrychlení algoritmu, pomocí efektivní vektorizace na GPU.

4.3 Momentum

SGD se často v praxi rozšiřuje o metodu Momentum. Základní myšlenka této metody je zrychlit konvergenci pomocí urychlení gradientů ve správném směru a zmírnit kličkování. To je uskutečněno přidáním části váhové změny z minulé iterace k současné iteraci. V rovnici 13 lze vidět

výpočet váhové změny, kterou je nutno si pamatovat do další iterace a v rovnici 14 je vidět aktualizace váhy. Hyperparametr metody je β, kterému se obecně říká momentum, stejně jako této metodě.

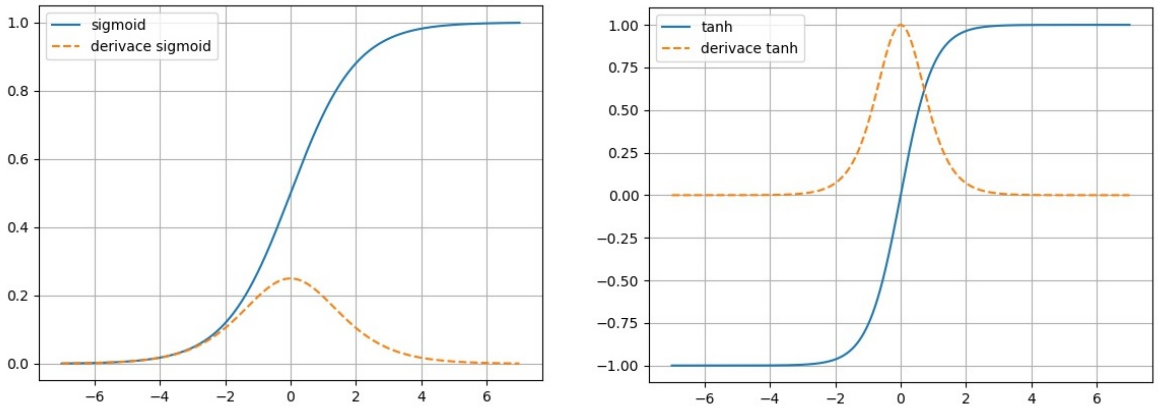
$$v = \beta v + \eta \frac{\partial L}{\partial w_{ij}^{l}(t)} \tag{13}$$

$$w_{ij}^{l}(t+1) = w_{ij}^{l}(t) - v \tag{14}$$

4.4 Problémy u aktivačních funkcí při trénování

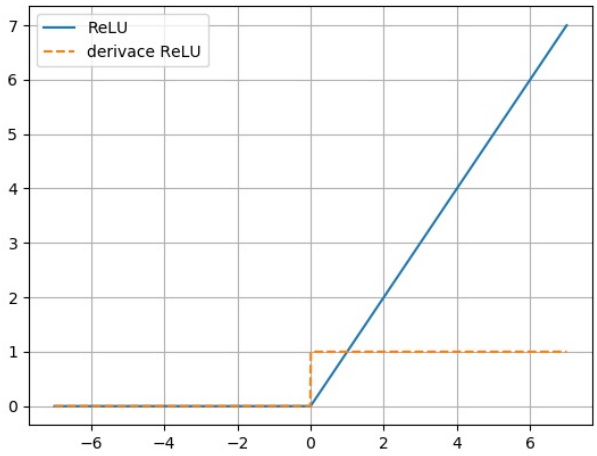
Pokud síť používá saturační aktivační funkci jako sigmoid nebo tanh, tak čím je hlubší síť, tím má větší tendenci trpět **problémem mizících gradientů (vanishing gradient problem)**. U tohoto problému se váhy v počátečních vrstvách aktualizují velmi pomalu, jelikož se při zpětném šíření gradient zmenšuje po každé vrstvě. To velice zpomalí trénování celé sítě, jelikož počáteční vrstvy se trénují pomalu a jejich následující vrstvy jsou závislé na jejich výstupech.

Pro výpočet gradientu chybové funkce je potřeba znát derivaci aktivační funkce (rovnice 11). Jak jde vidět na obrázku 4, derivace funkcí vrací hodnoty menší než 1 (zvláště funkce sigmoid, která vrací pouze hodnoty menší než 0,25). Gradienty se s využitím řetízkového pravidla počítají od poslední vrstvy k první, jelikož například k vypočítání zpětně šířené chyby v neuronech první vrstvy je potřeba znát všechny zpětně šířené chyby neuronů v druhé vrstvě. Tato zpětně šířená chyba se v každé vrstvě násobí výstupem derivace aktivační funkce a postupným násobením s těmito malými hodnotami se bude chyba zmenšovat. Tím pádem se taky zmenšují gradienty vah vypočtené z těchto chyb.



Obrázek 4: Sigmoid a tanh funkce a jejich derivace

Jak jde vidět na obrázku 5, tak použitím ReLU funkce se řeší problém mizících gradientů, jelikož derivace ReLU vrací buďto 0, nebo 1 (v sítí pořád může k tomuto problému dojít, ale ne kvůli ReLU).



Obrázek 5: ReLU a její derivace

Ale dochází k jinému problému, zvaný **Dying ReLU problem** (v některých případech ale tento problém nevadí, jelikož řídkost sítě přináší jisté výhody). V případě tohoto problému se daný neuron dostane do stavu, kde při dopředném průchodu sítí je vstupem do ReLU záporná hodnota v podstatě pro jakékoliv potenciální vstupy. Jelikož je derivace ReLU v tomto případě vždycky 0, tak gradient pro všechny váhy napojené na tento neuron bude taky 0. Váhy nejsou nadále aktualizovány a neuron nijak nepřispívá k síti, takže lze považovat za "mrtvý". Tomuto problému však jde do jisté míry předejít použitím alternativní varianty Leaky ReLU, která propouští malé hodnoty i při záporných vstupech (rovnice 15). Takhle se váhy mohou pomalu měnit, až do ReLU začnou vstupovat kladné hodnoty a neuron má šanci na zotavení.

$$f(x) = \begin{cases} x & \text{pokud } x > 0 \\ 0,01x & \text{jinak} \end{cases} \tag{15}$$

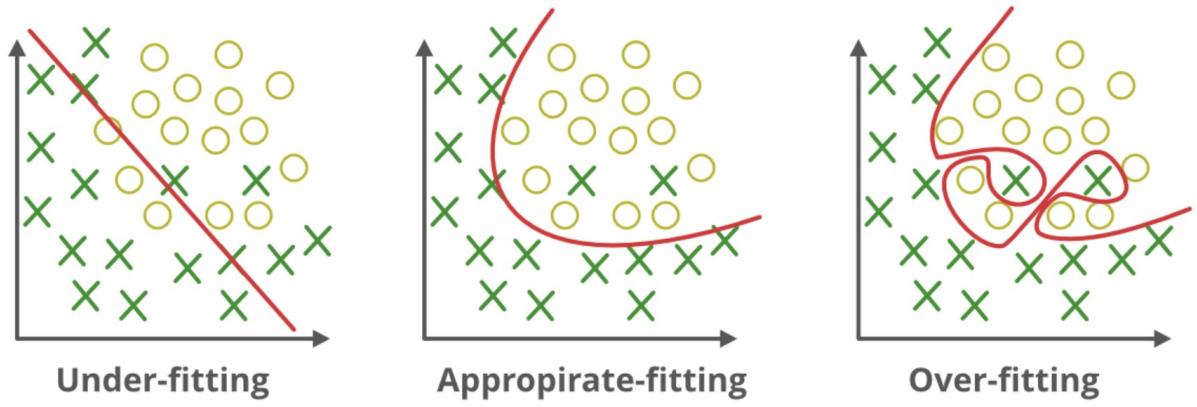
Při změně konstanty 0,01 na proměnnou, se stane proměnná trénovatelným parametrem sítě a této variantě aktivační funkce se říká Parametric ReLU.

5 Regularizace

Regularizační techniky slouží pro předcházení přetrénování. Při přetrénování (overfitting) ztrácí síť schopnost generalizace a to znamená že se model příliš nauč, neboli až zapamatuje trénovací sadu a poté vydává špatný výkon na dosud nespatřených datech (viz obrázek 6). V tom případě má natrénovaný model malou trénovací chybu a velkou generalizační chybu. To se u iteračních optimalizačních metod může stát při moc dlouhém trénování, čemuž předchází **brzké zastavení**, nebo při volbě příliš složitého modelu. Složitý model sice může být pro složitý problém nutný, ale zároveň je k natrénování takového modelu potřeba mnoho dat. Proto se složité modely penalizují, například pomocí **L2 regularizace**. Nebo lze také použít **augmentaci dat** pro rozšíření velikosti trénovací sady. Způsob augmentace dat se dost liší podle řešeného problému, proto sem se rozhodl popsat pouze jak je vhodné augmentovat data při trénování modelu pro rozpoznávaní obličejů (viz podkapitola 8.4.3). Další možností proti přetrénování je kombinace několika natrénovaných modelů a zprůměrování jejich predikcí. V neuronových sítích lze použít mnohem méně náročnou techniku zvanou **dropout** [29], kterou lze interpretovat jako kombinací několika "ztenčených" sítí. Všechny výše zmíněné techniky jsou v následujících podkapitolách podrobněji popsány a také jsou použity při experimentech.

Pro zkontrolování jestli jsou hyperparametry sítě správně nastavené (to zahrnuje i nastavení regularizačních metod) a nebo jestli dochází k přetrénování, se používá **křížová validace (cross-validation)**.

Opak přetrénovaní je podtrénovaní (underfitting), u kterého má model špatný výkon jak na trénovacích, tak na dosud nespatřených datech (obrázek 6). K tomu může u iteračních optimalizačních metod dojít při moc krátkém trénování (opět řešené brzkým zastavením) a nebo při volbě modelu, který nedokáže zachytit základní strukturu dat (např. lineární model pro lineárně neseparabilní data).



Obrázek 6: Vizualizace podtrénování a přetrénování na 2D datech [30]

5.1 L2 regularizace

L2 regularizace patří mezi regularizační techniky, které penalizují modely za jejich složitost a to přidáním regularizačního prvku k chybové funkci. Hyperparametr regularizace je λ, který se násobí s regularizačním prvkem a tak ovlivňuje jeho účinek. U L2 regularizace je regularizačním prvkem suma vah na druhou a nová chybová funkce tak vypadá:

$$L'(X, y, W) = L(X, y, W) + \frac{\lambda}{2} \sum w_i^2 \tag{16}$$

a aktualizace váhy u SGD vypadá:

$$w(t+1) = w(t) - \eta \frac{\partial L}{\partial w(t)} - \eta \lambda w(t). \tag{17}$$

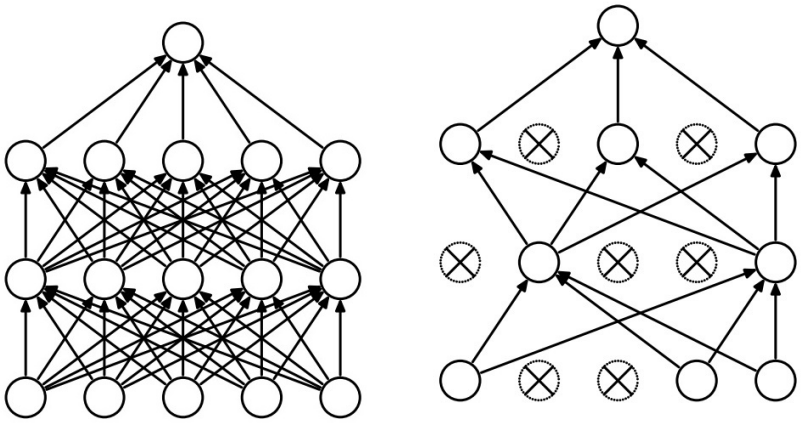
U každé optimalizační iterace jsou tak váhy nejen aktualizovány podle gradientu, ale navíc jsou úměrně utlumovány směrem k nule. Pokud si představíme trénovací data jako kombinaci signálu a šumu, tak pro optimalizaci chybové funkce bez regularizačního prvku zachytí dosti složitý model jak signál, který je důležitý pro generalizaci, tak i šum který vede k přetrénování. Intuice za L2 regularizací je taková, že při jejím použití bude pro model těžší naučit se šum v datech. Váhy jsou totiž neustále utlumovány a tím je také zvětšovaná chyba na trénovacích datech. Avšak důležité váhy, které přispívají k zachycení signálu se budou jednoduše zotavovat, aby tuto chybu zpátky zmenšily. Na druhou stranu váhy, které přispívají k zachycení konkrétních šumu v jednotlivých vzorkách, se tak jednoduše nezotaví.

5.2 Dropout

Přetrénování lze snížit kombinací několika modelů a při testování zprůměrovat jejich predikce. Tohle je však příliš časově náročné, zvláště u složitých modelů. Proto se používá *dropout*, kde výsledek metody by se dal interpretovat jako několik "ztenčených" sítí (které sdílí parametry) a při testování je aproximován jejich geometrický průměr, jednoduchým použitím jedné "neztenčené" sítě se zmenšenými váhami.

Dropout [29] je tedy regularizační technika, u které se během trénování v předem vybraných vrstvách, vypínají určitá procenta neuronů (obrázek 7). To znamená, že výstup vypnutých neuronů je 0 a gradienty vah napojené na dané neurony jsou taktéž 0. Neurony se vypínají náhodně a vybírají se znova u každé iterace.

Během běžného trénování může docházet mezi neuronama ke "společné adaptaci" (co-adaptation), přičemž se neurony mění tak, aby opravovaly chyby ostatních neuronů. Tyhle "společné adaptace" přidávají modelu složitost, která pomůže zmenšit trénovací chybu, ale nejsou dostatečně robustní, aby generalizovaly na dosud nespatřených datech. Autoři práce [29] spekulují, že dropout by měl této "společné adaptaci" předcházet, jelikož je přítomnost neuronů nespolehlivá.



(a) Klasická neuronová síť

(b) Neuronová síť s technikou dropout

Obrázek 7: Vizualizace techniky dropout [29]

V testovací fázi jsou již všechny neurony aktivní a proto se ve vrstvách, kde byl použitý dropout musí zmenšovat výstup neuronů, aby očekávány vstup do neuronů následující vrstvy nebyl příliš velký. Přesněji se výstup neuronů násobí s *1-p*, kde *p* je pravděpodobnost vypnutí neuronu. Další možností je zvětšit výstupy neuronů během trénování (vydělením s *p*) a během testování se již nemusí provádět žádné operace. Této funkčně ekvivalentní variantě se říká *inverted dropout*.

5.3 Brzké zastavení

Problém s trénováním neuronových sítí spočívá ve volbě správného počtu epoch. Příliš mnoho epoch může vést ke přetrénování, zatímco příliš málo může vést k nedostatečně natrénovanému modelu. Trénování by šlo zastavit, jakmile se přestane zmenšovat chyba na trénovací sadě, avšak v takovém případě je riziko přetrénování vysoké. Pro zastavení trénování mezi podtrénovaným a přetrénovaným modelem, jde použít metodu brzkého zastavení (early stopping).

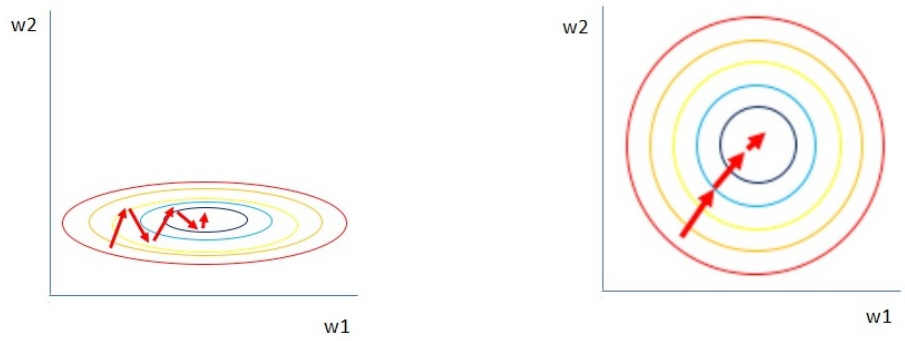
Data jsou v této metodě rozdělena na trénovací a validační data. Validační data, na kterých se síť neučí, slouží k ohodnocení modelu během trénování. Model může být ohodnocený chybovou funkcí, nebo jinými funkcemi pro ohodnocení kvality klasifikace. Tohle ohodnocení poskytuje informaci o tom, jak se síti daří na dosud nespatřených datech a díky tomu lze posoudit, jestli se síť stále učí a nebo se již přetrénovává. U základní varianty této metody se trénovaní zastaví, jakmile se výsledek monitorované funkce přestane zlepšovat, na předem daný počet epoch. Finálně vybraný model poté může mít parametry z epochy s nejlepším výsledkem.

6 Běžné praktiky

V této kapitole jsou popsány běžně používané praktiky, které umožní natrénovat hlubší a kvalitnější neuronové sítě. Všechny tyto praktiky jsou použity při experimentech.

6.1 Normalizace

Už je dlouho známo, že sítě rychleji konvergují, pokud mají normalizované vstupy [28]. Intuice za tímto tvrzením, lze znázornit na jednoduché síti s dvěma váhami $w1$ a $w2$. Pokud je vstup napojený na váhu $w1$ v řádu jednotek a druhý vstup napojený na váhu $w2$ v řádu stovek, tak bude gradient "kličkovat" po povrchu chybové funkce, jelikož $w2$ bude více přispívat k chybě. Nejenom to, ale taky se nepříznivě změní povrch chybové funkce. Protože i kdyby byly váhy nastaveny na hodnoty, při kterých obě váhy přispívají k chybě stejně, tak při aktualizaci vah to bude pro váhu $w2$ větší změna, jelikož je napojena na větší vstupy. Například na obrázku 8 lze vidět parabolický povrch chybové funkce, který se kvůli nenormalizovaným vstupům v jedné dimenzi zmenšil a gradient má tendenci se pohybovat více po souřadnici pro $w2$. A kvůli tomu déle trvá konvergence algoritmu.



(a) bez normalizace dat (b) s normalizovanými daty

Obrázek 8: Povrch chybové funkce [27]

U obrazů tento problém není tak výrazný, jelikož všechny pixely mohou nabývat minimální hodnoty 0 a maximální hodnoty 255. I přesto je normalizace vhodná a to zejména kvůli vycentrování dat kolem 0, což je známo pro zrychlení trénování sítí [28]. Běžně se také data transformují do menšího rozmezí, nebo se zmenšuje rozptyl dat, což vede k dodatečnému zlepšení stability při trénování.

Pro normalizaci dat lze použít **Min-Max normalizaci** (rovnice 18), která lineárně transformuje hodnoty do rozmezí <0, 1>. Častěji se však používá obdoba téhle metody, která transformuje hodnoty do rozmezí <-1, 1> a tak centruje data kolem 0. Před použitím je někdy vhodné se zbavit outlierů.

$$x' = \frac{x - min(x)}{max(x) - min(x)} \tag{18}$$

Další možností je **standardizace (z-skóre)**, která zajistí průměr 0 a směrodatnou odchylku 1:

$$x' = \frac{x - \mu}{\sigma}, \tag{19}$$

kde $\hat{x}$ je průměr a σ je směrodatná odchylka pro vektor hodnot x.

Při testování by měli testovací data projít stejnou transformací jako trénovací data. Proto se například při použití z-skóre ukládá průměr a rozptyl z trénovacích dat a pomocí nich jsou poté transformovány i testovací data.

6.2 Inicializace vah

Před trénováním sítě je potřeba nastavit parametry sítě na rozumné hodnoty. Biasy se většinou nastavují na hodnotu 0. U vah to tak jednoduché není a správná inicializace vah může výrazně zrychlit konvergenci sítě a při špatné inicializaci se může trénování zpomalit až k nepoužitelnosti.

Při nastavení všech vah na stejnou hodnotu je výstup všech neuronů ve vrstvě stejný a tím pádem budou stejně přispívat k chybě sítě. Z toho důvodu budou ve vrstvě stejné gradienty a váhy budou v jednotlivých iteracích při trénování stejné. Váhy jsou proto vybírány náhodně, aby byla rozbitá symetrie. Váhy jsou tedy vybrány z rovnoměrného nebo z normálního rozdělení. Avšak pokud je rozptyl v rozdělení nastavený na příliš malý nebo velký, tak hlubší sítě jsou mnohem náchylnější k mizícímu či explodujícímu gradientu, protože rozptyl hodnot ve vrstvách postupně klesá/stoupá každou vrstvou.

Nejdříve budou brány v potaz sítě s dříve nejpoužívanější aktivační funkcí hyperbolickou tangens (tanh). Při nastavení rozptylu vah na příliš velkou hodnotu, vzhledem k architektuře sítě, se bude při dopředném průchodu rozptyl hodnot vrstvy (před aktivační funkcí) zvětšovat až dojde k tomu, že se většina hodnot bude nacházet v "saturovaných" oblastí aktivační funkce (pro funkci tanh je to zhruba $|x| > 3$). V tom případě je derivace aktivační funkce skoro 0 a gradienty dané vrstvy budou tím pádem taky nabývat skoro nulových hodnot. Postupným zpětným šířením do méně hlubokých vrstev, však začne počet saturovaných neuronů klesnout a kvůli příliš velkým váhám dojde k podobnému problému jako u dopředného průchodu, akorát místo explodujících aktivací začnou explodovat gradienty. Zato při nastavení rozptylu vah na příliš malou hodnotu, vzhledem k architektuře sítě, se rozptyl výstupu vrstvy zmenší na hodnotu blížící se 0 (se střední hodnotou 0) a proto gradienty v dané vrstvě budou taky většinou skoro 0. Postupným zpětným šířením bude gradient čím dál více mizet.

Oba problémy naznačují, že by bylo optimální zachovat stejný rozptyl napříč všemi vrstvami, jak v dopředném průchodu pro aktivace, tak ve zpětném šíření pro gradienty. Právě s tímhle nápadem přišli v roce 2010 v práci [22], kde použili normalizovanou inicializaci, která je známá pod názvem **Xavier inicializace**:

$$\sigma = \sqrt{\frac{1}{N_{in}}} \quad (20) \qquad \sigma = \sqrt{\frac{1}{N_{out}}} \quad (21) \qquad \sigma = \sqrt{\frac{2}{N_{in} + N_{out}}}, \quad (22)$$

kde σ je směrodatná odchylka, N_{in} je počet neuronů ve vstupní vrstvě a N_{out} je počet neuronů ve výstupní vrstvě. Rovnice 20 je navržena pro zachování stejného rozptylu aktivací, 21 pro zachování stejného rozptylu gradientů a 22 je kompromis mezi 20 a 21. Při návrhu této inicializace byl předpokladem lineární režim aktivační funkce v době inicializace sítě. Proto je Xavier inicializace nejhodnější pro sítě s aktivační funkci tanh.

Dalo by se říct, že výhody ReLU funkce byly poprvé pořádně zvýrazněny v roce 2011 [23] a po jejím použití v průkopnické architektuře AlexNet [46] její popularita v hlubokých neuronových sítích pouze rostla. Proto je důležité vědět, jak inicializovat váhy i pro tyto sítě. Při inicializaci vah s konstantním rozptylem dochází v sítích s ReLU k podobným problémům, jako u sítí s funkci tanh. Avšak při nastavení rozptylu vah na příliš velkou hodnotu nedochází k saturaci neuronů, ale může dojít k *NaN* hodnotám, například v poslední vrstvě při počítaní exponenciálních funkcí v Softmax funkci. To se stane, jelikož ReLU není funkce shora omezená a v poslední vrstvě (před vstupem do Softmax) se budou nacházet obrovské hodnoty. Ale Xavier inicializaci taky není zrovna optimální použít, jelikož se u ní předpokládá že výstup každé vrstvy bude mít střední hodnotu 0 a to při použití ReLU funkce neplatí. Protože ReLU nastavuje záporné hodnoty na 0, tak by se po každé vrstvě zmenšoval rozptyl a ve velmi hluboké síti by takhle byly výstupy posledních vrstev takřka nulové. V roce 2015 byla v práci [24] použita pozměněná Xavier inicializace pro sítě s ReLU funkcemi, která je známá pod názvem **He inicializace**:

$$\sigma = \sqrt{\frac{2}{N_{in}}}. \quad (23)$$

V práci [24] také zmiňují, že v běžně používaných architekturách sítí může být jako jmenovatel použitý jak N_{in} tak i N_{out} a nedojde k explodujícím či mizícím aktivacím ani gradientům.

Bias se nastavuje v Xavier i He inicializaci na 0.

6.3 Batch normalizace

V průběhu trénování sítě jsou váhy v určité vrstvě aktualizovány s tím, že očekávají výstupy z minulé vrstvy s určitým rozdělením. Ale jelikož jsou zároveň aktualizovány váhy v předchozích vrstvách, tak je tohle rozdělení změněno. Tuto změnu nazývají v práci [25] *Internal Covariate Shift*. Problémem jevu *Internal Covariate Shift* je nutnost vah se neustále přizpůsobovat těmto změnám a takhle pořád "honí pohybující se cíl". Pro zmenšení tohoto jevu se používá **batch**

normalizace.

Výhody při použití batch normalizace:

- umožňuje použít vyšší learning rate
- dodá trochu regularizace pro lepší generalizaci, jelikož při batch normalizaci je vypočítán rozptyl a průměr pouze z dat v mini-batch (místo celého datasetu), takže tyto veličiny v sobě nesou jakýsi "šum" (z toho vyplývá čím menší mini-batch, tím větší regularizace)
- inicializace vah není až tak důležitá, hlavní je aby byla rozbitá symetrie (neinicializovat všechny váhy stejnou hodnotou)
- menší počet potřebných epoch pro natrénování sítě (i když se zároveň zvětší výpočetní náročnost)

V základní variantě Batch normalizace se pro každý mini-batch vypočítá průměr a rozptyl nezávisle v každém neuronu. Hodnoty pro jeden neuron se následně normalizují:

$$\mu_B = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \rightarrow \quad \sigma_B^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_B)^2 \quad \rightarrow \quad \hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \tag{24}$$

kde n je mini-batch velikost, x jsou hodnoty neuronu, μ_B je mini-batch průměr, σ_B^2 je mini-batch rozptyl, ϵ je konstanta aby nedošlo k dělení nulou a $\hat{x_i}$ je i-tá normalizovaná hodnota v mini-batch. V konvolučních sítích se μ a σ^2 počítá pouze pro každou příznakovou mapu, jelikož v ní všechny neurony sdílí stejné váhy.

Pouhá normalizace hodnot by omezila, co vrstva dokáže reprezentovat. Proto se dále normalizovaná hodnota transformuje pomocí parametrů γ a β, které definují nový průměr a směrodatnou odchylku:

$$y_i = \gamma\hat{x} + \beta, \tag{25}$$

kde y_i je již výstup po batch normalizaci pro i-tý vzorek v mini-batch. Oba parametry jsou součástí trénovatelných parametrů sítě pro každý neuron, nebo v případě konvolučních sítí pro každou příznakovou mapu.

Batch normalizace se většinou provádí na výstupech vrstvy před použitím aktivační funkce. V tom případě se nemusí používat bias, protože jeho efekt je zrušený při odečtení průměru při normalizaci. Role biasu je pak zastoupená parametrem γ.

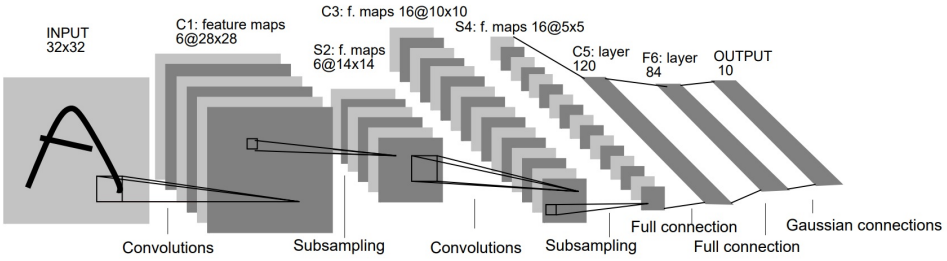
Během reálného testování jsou modelu prezentovány data po jednom vzorku a nelze vypočítat rozptyl nebo průměr na mini-batch. Proto je před testováním vypočítaný průměr a rozptyl ze všech trénovacích dat a pomocí těchto hodnot jsou pak normalizovaná testovací data. Další možností je během trénování počítat a uchovávat si exponenciální klouzavý průměr pro μ_B a σ_B^2 u každého neuronu.

I když snížení jevu *Internal Covariate Shift* byla motivace za metodou batch normalizace, tak novější práce [26] navrhuje, že "uhlazení" povrchu chybové funkce je hlavní důvod účinnosti batch normalizace a snížení jevu *Internal Covariate Shift* je pouze vedlejší účinek.

7 Architekrury

Původ architektury konvolučních neuronových sítí by se dal připsat práci [19], ve které je síť inspirovaná objevy ve vizuální kůře savců a je pojmenována **neocognitron**. V neocognitron jsou vrstvy složené buďto z S-buněk nebo C-buněk a tyto vrstvy se v případě S-buněk velmi podobají konvolučním vrstvám a v případě C-buněk pooling vrstvám. V roce 1980, kdy byla práce napsána, se však ještě pro trénovaní neuronových sítí nepoužívalo zpětné šíření. V roce 1989 [20] už byla použita konvoluční neuronová síť se zpětným šířením pro rozpoznání poštovních směrovacích čísel. Architektura byla pojmenována LeNet-1 a po několika vylepšeních byla v roce 1998 navržena její velice známá varianta **LeNet-5** [45] (obrázek 9), která byla zaměřena pro klasifikaci ručně psaných znaků. Ale i po zveřejnění LeNet-5 nebyly KNS moc používány, až dokud nedošlo k velkému průlomu v oblasti strojového učení v roce 2012 a to v podobě architektury **AlexNet** [46]. I když byla architektura oproti LeNet-5 vylepšena ve více směrech, tak pravděpodobně největší podíl v jejím úspěchu hrála vysoce optimalizovaná implementace na grafické karty (GPU). Využití grafických karet umožnilo trénování sítí na velkých datasetech a použití hlubších a tím pádem více výpočetně náročných architektur. Pro porovnání, AlexNet má přes 60 miliónů parametrů oproti 60 000 parametrů v LeNet-5.

Díky velkým veřejným datasetům (které jsou pro natrénovaní hlubokých sítí potřeba) jako ImageNet [8] a vzrůstu výpočetního výkonu grafických karet, bylo nyní možné natrénovat hluboké neuronové sítě, které výrazně překonávaly ostatní tradiční metody.



Obrázek 9: Architektura LeNet-5 [45]

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) je soutěž pro porovnání metod v detekci objektů a klasifikaci obrazů na podmnožině ImageNet databáze s více než milión obrazy a 1000 třídami [48]. V kategorii klasifikace obrazů jsou architektury hodnoceny na základě top-5 chybovosti, kde má síť za úkol identifikovat 5 objektů v obraze a pokud alespoň jeden z nich je správný, tak nedojde k žádné penalizaci. To se může zdát jako docela jednoduchý úkol, ale obrazy z ImageNet databáze v sobě obsahují mnoho objektů a pouze jeden dominantní objekt je považován za ten správný (obrázek 10).

Obrázek 10: Obraz z ImageNet databáze, na kterém je správné označení "ocelový buben" [48]

Soutěž byla pořádána každoročně od roku 2010 do 2017 a AlexNet byl použitý v kategorii klasifikaci obrazů v roce 2012. V soutěži obsadil 1. příčku s top-5 chybovostí (error rate) 15,3%, s obrovským náskokem v porovnání s druhou příčkou s top-5 chybovostí 26,2% [46]. Za zmínku stojí, že to byla jediná konvoluční neuronová sít v soutěži. Pochopitelně po takovém velkém úspěchu se upoutala na KNS pozornost a již v následujícím roce byla většina práci založena na KNS. Další úspěšné architektury, které v této soutěži výrazně posunuli laťku nahoru a zároveň přinesli nové důležité poznatky jsou **VGG** [50], **GoogLeNet** (Inception-v1) [47] a **ResNet** [51]. Avšak od roku 2015 již nedošlo k významným zlepšením, jelikož už není tolik místa pro zlepšení a pro další vývoj a porovnání architektur by byl zapotřebí složitější úkol. V posledním ročníku soutěže ILSVRC 2017 měla nejlepší síť top-5 chybovost 2,25% [49].

V tabulce 1 jsou pro přehlednost vypsány sítě a jejich vlastnosti ze soutěže ILSVRC, v kategorii klasifikace obrazů. Všechny chybovosti jsou výsledkem ensemble několika modelů s danou architekturou. Autoři daných prací také každý jinak rozšířili předem daný ImageNet dataset, pomocí augmentace dat, jako další proti opatření proti přetrénování. Tyto metody augmentace dat se dost liší oproti augmentaci dat při práci s datasety obličejů, takže nebudou rozebírány. Více o augmentaci dat u rozpoznávání obličejů v podkapitole 8.4.3.

	top-5 chybovost [%]	Počet vrstev	Počet parametrů [mil]
AlexNet (2012)	15,3	8	60
VGG (2014)	7,32	16/19	138/144
GoogleNet (2014)	6,67	22	6,8
ResNet (2015)	3,57	152*	60

Tabulka 1: Úspěšné architektury ze soutěže ILSVRC v kategorii klasifikace obrazů
*152 vrstev má nejhlubší použitá síť v ensemble, ale jsou použity i méně hluboké sítě

Všechny architektury v tabulce 1 jsou následně podrobně popsány a to hlavně z hlediska jak

ovlivnili vývoj konvolučních sítí jako takových. Také je s nimi experimentováno v podkapitole 8.6.1. ResNets a Inception sítě by se daly považovat za state-of-the-art i po mnoha letech od jejich vynalezení, díky jejím dalším vylepšeným variantám. Několik z těchto variant je taky dále stručně popsáno.

7.1 Alexnet

AlexNet byl představen v roce 2012 v práci [46] a byl pojmenován po jednom z autorů Alexovi Krizhevsky. Návrh architektury je i přes větší hloubku a šířku dost podobný architektuře LeNet-5, ale jsou v ní skryté důležité rozdíly. V té době často používanou aktivační funkci tanh, která byla také použita v LeNet-5, nahradili jako jedni z prvních za aktivační funkci ReLU. V této práci hlásí jako hlavní motivaci pro použití ReLU její schopnost zrychlit konvergenci sítě. Average pooling vrstvy jsou nahrazené za max pooling vrstvy, u kterých navíc autoři zjistili, že překrývající se podoblasti obecně trochu pomáhají proti přetrénování a také jim na ImageNet datasetu mírně zmenší chybovost. V tabulce 2 je podrobně popsána celá architektura, kde u všech konvolučních a plně propojených vrstev je aktivační funkce ReLU, až na poslední plně propojenou vrstvu s funkcí Softmax.

Vrstva	**Počet filtrů**	**Velikost filtrů**	**Velikost výstupu**	**Krok**	**Zero-padding**	**Počet parametrů**
Vstupní			$227 \times 227 \times 3$			
Konvoluční 1.	96	11×11	$55 \times 55 \times 96$	4	Ne	34 944
Max pooling		3×3	$27 \times 27 \times 96$	2	Ano	
Konvoluční 2.	256	5×5	$27 \times 27 \times 256$	1	Ano	614 656
Max pooling		3×3	$13 \times 13 \times 256$	2	Ano	
Konvoluční 3.	384	3×3	$13 \times 13 \times 384$	1	Ano	885 120
Konvoluční 4.	384	3×3	$13 \times 13 \times 384$	1	Ano	1 327 488
Konvoluční 5.	256	3×3	$13 \times 13 \times 256$	1	Ano	884 992
Max pooling		3×3	$6 \times 6 \times 256$	2	Ano	
Plně propojená 1.			4096			37 752 832
Dropout (0,5)			4096			
Plně propojená 2.			4096			16 781 312
Dropout (0,5)			4096			
Výstupní			1000			4 097 000

Tabulka 2: AlexNet architektura

Jak jde vidět, tak většina parametrů se nachází v plně propojených vrstvách, přesněji 3 747 200 (6%) v konvolučních a 58 631 144 (94%) v plně propojených. Novinka v této architektuře je regularizační technika **dropout** u 1. a 2. plně propojené vrstvy. U těchto vrstev je použita právě proto, že se zde nachází velký počet parametrů, kvůli kterým je síť náchylná k přetrénování. Na ImageNet datasetu se bez metody dropout síť podstatně přetrénovávala, ale s jejím použitím

se počet potřebných iterací ke konvergenci zdvojnásobil [46]. Dropout a pojem přetrénování jsou podrobněji popsány v kapitole 5.

Dále byla v některých konvolučních vrstvách použitá Local Response Normalizace (LRN), ale ta již byla nahrazena jinými technikami a například u VGG architektury (která je probrána v další podkapitole) zjistili, že jim tato normalizace nijak nezlepšuje výkon sítě a pouze zabírá čas a paměť. Z toho důvodu nebude LRN dále rozebírána.

7.2 VGG

VGG síť [50] byla předložena v ILSVRC 2014 a v kategorii klasifikace obrazů obsadila druhé místo s top-5 chybovostí 7,32%. Architektura je stále navržena podobně jako LeNet a AlexNet, ale je mnohem hlubší a používá pouze malé 3×3 filtry.

V práci vytvořili obecnou architekturu s 5 blokama, ve kterých se nachází konvoluční vrstvy s ReLU. Do těchto bloků postupně přidávali další konvoluční vrstvy a zvětšovali tak hloubku sítě z 11 až do 19 vrstev a pozorovali že se klasifikační chyba postupně zmenšuje. Nejlepších výsledků dosáhli architektury s 16 a 19 vrstvami, které jsou známy pod názvy VGG-16 a VGG-19. I když se architektura oproti AlexNetu značně prohloubila, tak šířka sítě zůstala srovnatelná. Charakteristickou vlastností VGG sítí, kterou také adoptovali některé ostatní architektury, je po každém půlení velikosti příznakových map také zdvojnásobit počet filtrů.

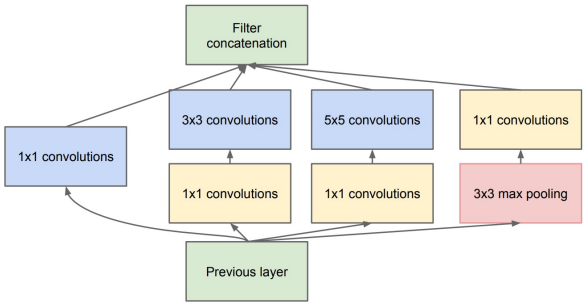
Rozhodnutí použít malé 3×3 filtry oproti větším filtrům jako 11×11, 7×7 nebo 5×5, které byly použity v dosavadních architekturách, přináší se sebou dvě hlavní výhody. Vrstvy s těmito většími filtry se dají vzhledem k recepčnímu poli (receptive field) filtru, nahradit umístěním několika vrstev s 3×3 filtry za sebou. Například 11×11 vrstva jde nahradit pěti 3×3 vrstvami. Při zpracování jedné příznakové mapy s 11×11 filtrem je potřeba $11 * 11 = 121$ parametrů a s pěti 3×3 filtry je potřeba pouze $5 * (3 * 3) = 45$, což je o 63% méně. První výhodou je tedy snížení počtu trénovatelných parametrů. Druhá výhoda je více diskriminační rozhodovací funkce, jelikož má síť více konvolučních vrstev a v každé je použitá nelineární funkce (ReLU).

V takto hlubokých sítích je důležitá správná inicializace vah, viz podkapitola 6.2. I přesto v této práci inicializovali sítě z normálního rozdělení s konstantním rozptylem 0,001. To by pravděpodobně vedlo k mizícím gradientům a proto tomuto problému předešli natrénováním menší sítě a poté váhami z této sítě inicializovali některé vrstvy jejich hlubší sítě. V práci také poznamenali, že v době jejího předložení nevěděli o Xavier inicializaci (a He inicializace ještě neexistovala).

7.3 Inception sítě

Dosavadní úspěšné sítě byly vytvořeny pouze ze sériově naskládaných vrstev na sebe a zlepšení se víceméně provádělo laděním hyperparametrů, změnou kombinací vrstev a prohlubování sítě. V práci [47] představili takzvaný **Inception modul** (obrázek 11), ve kterém jsou větvě vrstev zapojené paralelně a poté jsou jejich výsledky spojené. V jedné vrstvě se takhle na ten

stejný vstup provádí pooling a konvoluce s 1×1, 3×3 a 5×5 filtrem. Konvoluční filtry 1×1 jsou přidané před výpočetně náročné 3×3 a 5×5 filtry pro redukci hloubky minulé vrstvy. Bylo zjištěno, že tímto se výrazně sníží výpočetní náročnost s minimálním snížením výkonu [47]. Před 3×3 max pooling není přidán 1×1 filtr, jelikož pooling není výpočetně náročný. Ale zato musí být přidán po pooling vrstvě, protože pooling zachovává hloubku předešlé vrstvy a to by po spojení všech výsledků na konci modulu znamenalo příliš hluboký výstup. Obecně se sítím složených z těchto modulů říkají Inception sítě.



Obrázek 11: Inception modul [47]

Jednu verzi takové Inception sítě v práci pojmenovali **GoogLeNet** (později pojmenovaná také Inception-v1), kterou použili v ILSVRC 2014 a v kategorii klasifikace obrazů obsadili první místo s top-5 chybovosti 6,67%. Oproti VGG, který se umístil jako druhý, má GoogleNet přibližně 20x méně parametrů.

V tabulce 3 je popsaná architektura, kde po každé konvoluci je použitá aktivační funkce ReLU a na konci sítě je funkce Softmax. 3×3 *reduce* a 5×5 *reduce* sloupce znamenají počet 1×1 filtrů před 3×3 a 5×5 konvolucemi. 1×1 *pool* znamená počet 1×1 filtrů po max pooling. Zajímavý je konec sítě, kde je pouze jedna plně propojená vrstva a před ní average pooling vrstva. Tento nápad pochází z práce [52], kde navrhli strategii **global average pooling**, ve které se úplně nahradí plně propojené vrstvy za jednu average pooling vrstvu. V této strategii má poslední konvoluční vrstva stejný počet příznakových map jako je tříd. Na to je napojená average pooling vrstva, kde je velikost oblasti stejně velká jako velikost příznakové mapy. Takhle je vytvořen průměr každé příznakové mapy a výstupem je vektor, který je vložen přímo do Softmax funkce. GoogLeNet se i přesto od originální strategie o trochu liší a přidává po average pooling vrstvě jednu plně propojenou vrstvu, která zjednodušší ladění sítě při výběru jiných tříd na klasifikaci. Výhodou odstranění několika plně propojených vrstev je i odstranění náchylnosti sítě k přetrénovaní, ale i přesto autoři uznali použití techniky dropout u předposlední vrstvy jako nezbytné.

Další zajímavostí je použití dvou pomocných klasifikátorů, které byli v originální práci zamýšleny pro redukci problému mizícího gradientu, avšak v novější práci je místo toho pozorováno, že pomocné klasifikátory přidávají regularizaci [31]. Tyto klasifikátory jsou menší konvoluční neuronové sítě, které jsou napojeny na výstup inception modulů 4a a 4d. Během trénování je

chyba pomocných klasifikátorů (s váhou 0,3) přičtena k hlavní chybě sítě. Při testování jsou pomocné klasifikátory ignorovány. Avšak pozdější experimenty ukázaly, že efekt těchto pomocných klasifikátorů je minimální (kolem 0,5%) a při použití jednoho klasifikátoru, je dosažen stejný efekt jako při obou [47].

Typ	**Velikost filtrů**	**Krok**	**Velikost výstupu**	1×1	3×3 **reduce**	3×3	5×5 **reduce**	5×5	1×1 **pool**
Vstupní			$224 \times 224 \times 3$						
Konvoluční	7×7	2	$112 \times 112 \times 64$						
Max pooling	3×3	2	$56 \times 56 \times 64$						
Konvoluční	1×1	1	$56 \times 56 \times 64$						
Konvoluční	3×3	1	$56 \times 56 \times 192$						
Max pooling	3×3	2	$28 \times 28 \times 192$						
Inception (3a)			$28 \times 28 \times 256$	64	96	128	16	32	32
Inception (3b)			$28 \times 28 \times 480$	128	128	192	32	96	64
Max pooling	3×3	2	$14 \times 14 \times 480$						
Inception (4a)			$14 \times 14 \times 512$	192	96	208	16	48	64
Inception (4b)			$14 \times 14 \times 512$	160	112	224	24	64	64
Inception (4c)			$14 \times 14 \times 512$	128	128	256	24	64	64
Inception (4d)			$14 \times 14 \times 528$	112	144	288	32	64	64
Inception (4e)			$14 \times 14 \times 832$	256	160	320	32	128	128
Max pooling	3×3	2	$7 \times 7 \times 832$						
Inception (5a)			$7 \times 7 \times 832$	256	160	320	32	128	128
Inception (5b)			$7 \times 7 \times 1024$	384	192	384	48	128	128
Average pooling	7×7	1	$1 \times 1 \times 1024$						
Dropout (0,4)			$1 \times 1 \times 1024$						
Výstupní			1000						

Tabulka 3: GoogLeNet architektura. Inception modul je zobrazen na obrázku 11

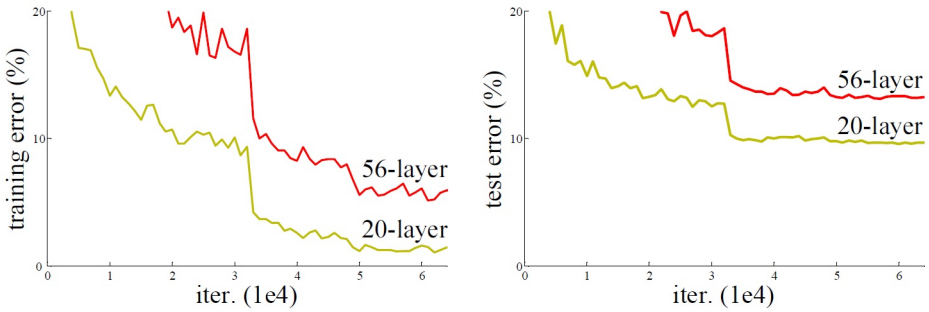
Další varianty Inception architektur:

- **BN-Inception**, taky nazývaná **Inception-v2**, byla zveřejněna ve stejné práci jako Batch normalizace [25]. Změna oproti architektuře GoogLeNet (Inception-v1) je právě přidání Batch normalizace a díky dodatečné regularizaci je odstraněný dropout v předposlední vrstvě. Další změnou je faktorizace 5×5 konvolučních filtrů v inception modulech na dva 3×3 filtry, stejně jako v VGG sítích.
- V architektuře **Inception-v3** [31] je také dále aplikovaná faktorizace na první 7×7 konvoluční filtry do třech 3×3 filtrů a v některých modulech je přidaná faktorizace na asymetrické konvoluční filtry, například 3×3 na 1×3 a 3×1 filtry. Dále jsou pro zmenšení příznakových map přidány speciální inception moduly, namísto dosud používaných max pool vrstev.
- **Inception-v4** [32] je hlubší a širší (více filtrů v jednotlivých vrstvách) varianta Inception-v3 s pozměněným začátkem sítě před inception modulama.

- **Inception-ResNet** [32] dále navazuje s přidáním ResNet zkratek uvnitř upravených inception modulů. Princip ResNet zkratek je podrobně vysvětlen v další podkapitole.

7.4 ResNet

Jak bylo ukázáno u architektury GoogLeNet a VGG, tak hloubka sítě hraje velkou roli ve výkonu sítě. Problémem však je, že po určité hloubce se začně přesnost sítě zhoršovat. Jedním z důvodů by mohlo být zvětšování problému mizících či explodujících gradientů. Tyto problémy ale byli z větší části vyřešeny s normalizovanou inicializaci vah [24] a batch normalizaci [25] (které mimochodem v době VGG a GoogLeNetu zatím neexistovali). Dalším problémem by u hlubokých sítí mohlo být přetrénovávání, kvůli jejich potřebě pro komplexnější problém nebo více dat. Jak je ale vidět na obrázku 12, tak ani to není problém, protože má hlubší síť horší nejenom testovací chybu, ale také trénovací chybu. I přesto se však po pár desítkách vrstev začne přesnost sítě zhoršovat a tomu v práci [51] říkají degradační problém. V této práci navrhují proč dochází k degradačnímu problému a zároveň nabízí jeho řešení s pomocí Residual Networks.

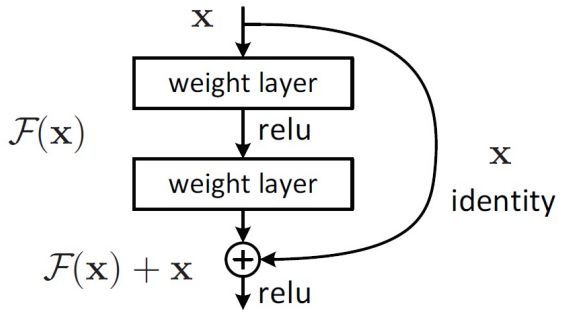


Obrázek 12: Degradační problém u prostých sítí na datasetu CIFAR-10. Podobný jev v práci také pozorovali na datasetu ImageNet [51]

Residual Network (ResNet) byla použita v ILSVRC 2015, kde v kategorii klasifikace obrazů obsadila první místo s top-5 chybovosti 3,57%. V ensemble modelů těchto sítí byly použité sítě s hloubkou až 152 vrstev.

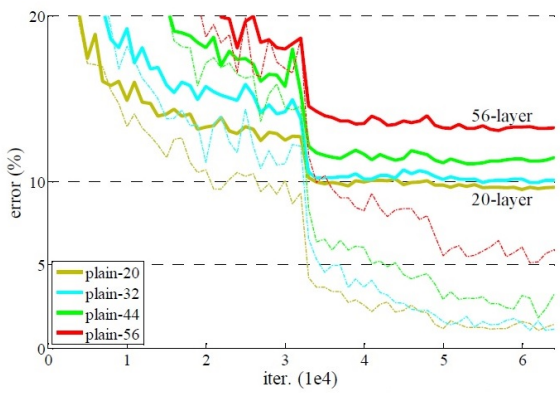
Nechť $H(x)$ je funkce kterou chceme aproximovat. Tato aproximace bude uskutečněna několika za sebou naskládaných nelineárních vrstev a x je vstup do první z těchto vrstev. Pokud předpokládáme, že $H(x)$ lze aproximovat, pak lze aproximovat také $H(x) - x$. Takže naskládané vrstvy budou aproximovat takzvanou reziduální funkci $F(x) = H(x) - x$ a původní funkce lze vyjádřit $H(x) = F(x) + x$. Žádoucí mapování $H(x)$ lze v síti realizovat pomocí přímého spojení neboli zkratky, jak je znázorněno na obrázku 13. Celému propojení od vstupu x až po výstup $H(x)$ se říká **reziduální blok**. Tento nápad vychází z faktu, že pokud bychom chtěli pouze zvýšit hloubku sítě a přitom nezhoršit přesnost, tak stačí přidat vrstvy, které pouze posílají stejné hodnoty dále. Jinými slovy by hlubší síť měla mít vždycky pouze stejnou nebo vyšší přesnost, než její méně hluboký protějšek, pokud je schopná svoje hlubší vrstvy nastavit na $g(x) = x$. To

se ale v prostých sítích (tak říkají v práci sítím bez reziduálních bloku) neděje. Takové vrstvy by se dali realizovat pomocí reziduálních bloků s jednoduchým nastavením vah tak, aby $F(x) = 0$ a zbude pouze $H(x) = x$. Pokud by však reziduální funkce $F(x)$ dokázala zlepšit přesnost sítě, tak by teoreticky mohla v reziduálním bloku přispívat k výstupu. V reálných případech sice není pravděpodobné že by $H(x) = x$ bylo optimální, ale ukázalo se, že tohle mapování pomáhá vytvořit podmínky, ve kterých se váhy v reziduálním bloku jednodušeji optimalizují. Z toho lze usoudit, že sítě bez reziduálních bloku mají možná problém aproximovat $H(x) = x$ v několika za sebou naskládaných nelineárních vrstvách.

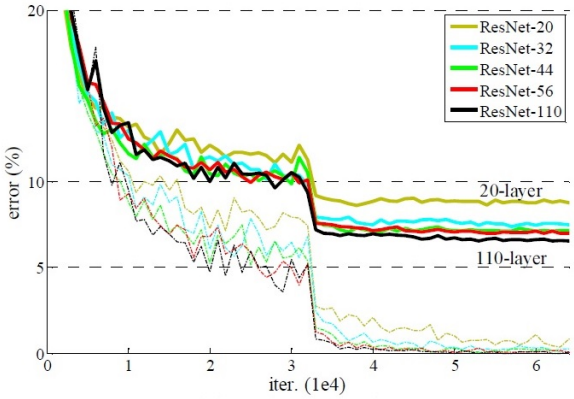


Obrázek 13: Reziduální blok [51]

Dimenze vstupu reziduálního bloku a jeho výstupu musí být stejné. Pokud se tedy změní počet příznakových map, nebo se zmenší příznakové mapy skrze pooling (nebo konvolucí s větším krokem než 1), tak se musí buďto vstup a výstup doplnit nulami a nebo použít 1×1 konvoluční filtry na vstup tak, aby se dimenze shodovaly. Při doplnění vstupu nulama má síť o trošku horší přesnost (o 0,3% horší top-5 chybovost na ImageNet), ale zato zkratky v reziduálních blokách nepřidávají žádné parametry nebo výpočetní náklady navíc (až na pár součtů vektorů). To je taky vhodné při porovnávání ResNets oproti prostým sítím se stejným počtem parametrů, hloubkou a šířkou. Takové porovnání jde vidět na obrázku 14, kde jde vidět zhoršující se prosté sítě při každém zvětšení hloubky a naopak zlepšující se ResNets. Podobný jev byl v práci pozorován na více datasetech. Na obrázku 15 jde dokonce vidět ResNet s 1202 vrstvami, která je sice o trochu horší než ResNet s 110 vrstvami, ale trénovací chybu mají přibližně stejnou, takže jde pravděpodobně o přetrénování než o degradační problém. Všechny ResNets i prosté sítě mají inicializované váhy pomocí He inicializace a po každé konvoluční vrstvě je aplikovaná Batch normalizace. Aktivační funkce ReLU je aplikovaná po každé Batch normalizaci, s výjimkou poslední Batch normalizace v reziduálním bloku, kde je ReLU použitá až po sečtení příznakových map jak je znázorněno na obrázku 13.

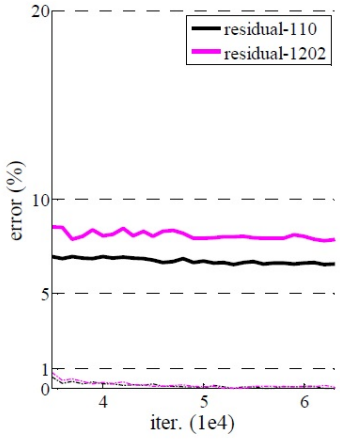


(a) Prosté sítě



(b) ResNets

Obrázek 14: Průběh trénování na datasetu CIFAR-10. Přerušovaná čára značí trénovací chybu a tučná čára značí testovací chybu [51]



Obrázek 15: Porovnání ResNet s 1202 a 110 vrstvami na datasetu CIFAR-10. Přerušovaná čára značí trénovací chybu a tučná čára značí testovací chybu [51]

Další varianty ResNet architektur:

- **Pre-activation ResNet** [33] má přemístěnou aktivační funkci ReLU před sčítání příznakových map a pořadí operací uvnitř bloků je změněno na BN (Batch normalizace)->ReLU->Konvoluce->BN->ReLU->Konvoluce.
- **ResNeXt** [34] podobně jako Inception-ResNet kombinuje nápady z inception modulu a ResNet bloku. ResNeXt blok se skládá z několika větví, avšak oproti inception modulu jsou všechny větve totožné a na konci bloku se příznakové mapy sčítají, místo spojení dohromady. Zkratka je použita stejně jako u klasického reziduálního bloku.

8 Experimenty

Celá tato kapitola je věnována praktické části diplomové práce.

8.1 Knihovny

Implementace proběhla v jazyce Python verzi 3.6. Tento programovací jazyk je často používán v komunitě strojového učení a má široký výběr knihoven, které umožnují efektivní manipulaci dat a nabízí mnoho užitečných algoritmů. Použité knihovny jsou shrnuty v tabulce 4.

Knihovny pro hluboké učení:

- **Keras** [55] je knihovna vytvořena zaměstnancem Googlu Françoisem Cholletem a napsaná v Pythonu s otevřeným zdrojovým kódem. První verze byla zveřejněna v roce 2015. Knihovna nabízí vysoko-úrovňové stavební bloky pro vývoj modelů hlubokého učení. To zahrnuje například aktivační funkce, chybové funkce, optimalizátory a vrstvy používány v konvolučních a rekurentních sítích. Nezabývá se tedy nízko-úrovňovými operacemi, jako jsou například tenzorové součiny a proto se spoléhá na jiné optimalizované knihovny, které ji slouží jako backend. Dostupné backend implementace pro Keras jsou na knihovny **TensorFlow**, Theano a CNTK.

- **TensorFlow** [42] je matematická knihovna s otevřeným zdrojovým kódem, vytvořena výzkumným týmem Google Brain a používána pro strojové učení. První verze byla zveřejněna na konci roku 2015 a od té doby popularita knihovny prudce rostla. S tím také rostla její popularita jako backend pro Keras. TensorFlow se tak stal zdaleka nejpoužívanějším backendem a z toho důvodu bylo zveřejnění Keras 2.3 (Září 2019) poslední verzí Kerasu jakožto samostatné multi-backend knihovny. Na druhou stranu se z zveřejněním verze TensorFlow 2.0 stal Keras její oficiální součástí a všechny budoucí aktualizace budou v rámci balíčku *tf.keras*.

- **Dlib** [54] je knihovna vytvořena Davisem E. Kingem a napsaná v C++ s otevřeným zdrojovým kódem. Zaměřuje se zejména na algoritmy pro strojové učení a zpracování obrazu. Knihovna je vyvíjena od roku 2002 a do dneška vychází časté aktualizace. Využívá se v široké škále oblastí včetně robotiky, vestavěných zařízení, mobilních telefonů a ve vysoce výkonných výpočetních prostředích.

- Další často používané knihovny v oblasti hlubokého učení jsou například **PyTorch** a **Caffe**.

Knihovny pro obecnější použití:

- **Open source Computer Vision (OpenCV)** [43] je knihovna napsána a vyvíjená v C++ s otevřeným zdrojovým kódem. Zaměřuje se zejména na počítačové vidění a zpracování

obrazu v reálnem čase. V roce 1999 byla zveřejněna první alfa verze firmou Intel a od té doby proběhlo mnoho vylepšení a stále vychází pravidelně nové verze.

- **Scikit-learn** [44] je knihovna zaměřená na strojové učení, která je napsaná pro programovací jazyk Python. Nabízí širokou škálu jednoduše používaných algoritmů pro klasifikaci, regresi a shlukování. Dále také nabízí jednoduchou manipulaci s daty v rámci strojového učení a předzpracování dat.

- **NumPy** je základní knihovna pro vědecké výpočty v programovacím jazyce Python, jejíž počáteční vydání sahá až do roku 1995. Mimo jiné nabízí infrastrukturu pro práci s vektory, maticemi a vícerozměrnými poli.

Knihovna	Verze	Poznámky
Tensorflow	2.1.0	Softwarové požadavky pro GPU podporu: CUDA 10.1 a cuDNN 7.6. Součástí této knihovny je také balíček *tf.keras*.
OpenCV-Python	4.2.0	Knihovna je v této práci použita pro načítání obrázků, zarovnání tváří a augmentaci dat. Při použití v jazyce Python automaticky převádí data do NumPy struktur.
Scikit-learn	0.22.2	
NumPy	1.18.2	

Tabulka 4: Použité knihovny

8.2 Implementace

V této kapitole jsou ukazány důležité úryvky z implementace s pomocí knihovny Tensorflow, především s balíčkem *tf.keras*.

Pro vytvoření modelu je využitá **functional API**, která je mnohem flexibilnější než *tf.keras.Sequential* API. S functional API lze vytvořit například síť s nelineární topologií, jako je Inception síť. Ve výpisu 1 jde vidět implementace inception modulu z obrázku 11. Parametry této metody jsou výstup minulé vrstvy a počet filtrů pro každou konvoluční vrstvu uvnitř modulu. Výstupem jsou již "za sebou" spojené příznakové mapy, z výstupu všech 4 větví uvnitř modulu.

```
def inception_module(inpt, f1, f2_reduce, f2, f3_reduce, f3, f4):
    conv_1x1 = Conv2D(f1, kernel_size=(1, 1), activation='relu')(inpt)

    conv_3x3_ = Conv2D(f2_reduce, kernel_size=(1, 1), activation='relu')(inpt)
    conv_3x3 = Conv2D(f2, kernel_size=(3, 3), padding='same', activation='relu')
        (conv_3x3)
```

```
    conv_5x5 = Conv2D(f3_reduce, kernel_size=(1, 1), activation='relu')(inpt)
    conv_5x5 = Conv2D(f3, kernel_size=(5, 5), padding='same', activation='relu')
        (conv_5x5)

    pool_proj = MaxPooling2D((3, 3), strides=(1, 1), padding='same')(inpt)
    pool_proj = Conv2D(f4, kernel_size=(1, 1), activation='relu')(pool_proj)

    module_out = concatenate([conv_1x1, conv_3x3, conv_5x5, pool_proj], axis=3)
    return module_out
```

Výpis 1: Inception modul s tf.keras functional API

V kapitole 8.6.2 je experimentováno s chybovými funkcemi Center loss [15] a ArcFace [16]. Tyto chybové funkce nejsou součástí knihovny Tensorflow, takže jsem si je musel implementovat sám.

Jelikož je pro výpočet **Center loss** (rovnice 9) nutné si pamatovat centra tříd, tak je tato chybová funkce implementovaná jako vrstva a centra jsou uchovávané jako váhy. Pro vytvoření vlastní vrstvy se rozšiřuje třída *tf.keras.Layer* a je potřeba implementovat metody *___init___*, *build* a *call*. V *___init___* a *build* se inicializují proměnné, avšak metodě *build* je již známá velikost vstupu do vrstvy. Ve výpisu 2 jsou v metodě *build* inicializované váhy neboli centra tříd o velikosti $počet\,tříd \times velikost\,vektoru\,příznaků$. Váhy jsou nastavené jako netrénovatelné, aby nebyly měněny během zpětného šíření. Místo toho jsou centra aktualizována v metodě *call* (výpis 3), kde jsou vstupem vektory příznaků a označení tříd ve formátu *kódu 1 z N* (počet vektorů a označení záleží na velikosti mini-batch). Aktualizace center se provádí s $\Delta centers$ ($delta\_centers$), pro které je nejprve nutno vypočítat rozdíl mezi vektory příznaků a jejich příslušnými třídními centry, tyto rozdíly ($delta\_batch$) jsou poté v rámci stejných tříd sečteny ($summed\_centers$) a ještě je pro ně vypočítán průměr ($averaged\_centers$). Hyperparametr α ($self.alpha$) je nastavený při vytvoření vrstvy a udává rychlost aktualizace center. Ve vrstvě nejsou jenom uchovávány třídní centra, ale výstupem už je rovnou chyba pro každý vektor příznaků.

```
def build(self, input_shape):
        self.class_centers = self.add_weight(name='class_centers',
                                    shape=(input_shape[1][1], input_shape[0][1]),
                                    initializer='uniform',
                                    trainable=False)
        super(CenterLossLayer, self).build(input_shape)
```

Výpis 2: Inicializace třídních center

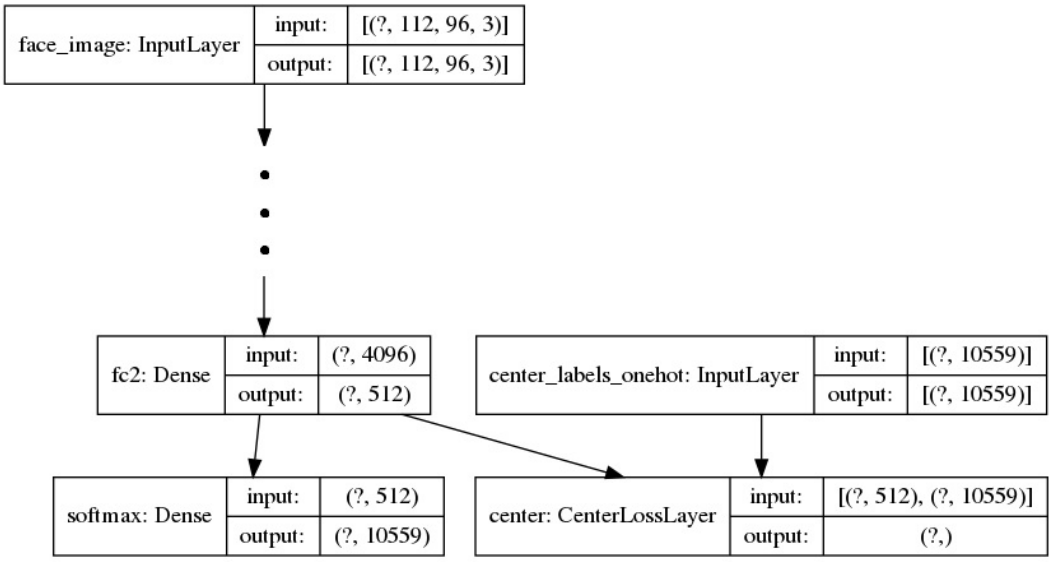
```
def call(self, x, mask=None):
        embeddings, one_hots = x

        centers_batch = K.dot(one_hots, self.class_centers)
        delta_batch = centers_batch - embeddings

        summed_centers = K.dot(K.transpose(one_hots), delta_batch)

        counts = K.sum(K.transpose(one_hots), axis=1, keepdims=True)
        class_zeros = K.zeros_like(counts) # maska pouzita pro prevenci proti
            deleni 0
        averaged_centers = K.switch(K.equal(counts, class_zeros),
            summed_centers / (counts+1), summed_centers / counts)

        delta_centers = self.alpha * averaged_centers
        updated_class_centers = self.class_centers - delta_centers
        self.add_update((self.class_centers, updated_class_centers))

        losses = K.sum(K.square(embeddings - centers_batch), axis=1)
        return losses
```

Výpis 3: Aktualizace třídních center a výpočet chyb

Center loss je při trénování použita společně se Softmax loss. Síť má tím pádem dva výstupy, jak je ukázáno na obrázku 16. Při konfiguraci modelu pro trénování (výpis 4), se pro výpočet Softmax loss (rovnice 6) nastavuje *tf.keras.losses.CategoricalCrossentropy()* a pro výpočet Center loss se nastavuje *my_losses.batch_mean()*. Jelikož je výstup z vrstvy CenterLossLayer už chyba pro každý vektor příznaku v mini-batch, tak v *my_losses.batch_mean()* se vypočte pouze průměr těchto chyb. Chyby obou funkcí jsou sečteny a k jejich vyvážení je použita hodnota λ (center_lambda).

```
model.compile(loss={'softmax': tf.keras.losses.CategoricalCrossentropy(), '
    center': my_losses.batch_mean()}, loss_weights={'softmax': 1.0, 'center':
    center_lamdba}, ...)
```

Výpis 4: Nastavení chybových funkcí při konfiguraci modelu



Obrázek 16: Zakončení sítě s Center loss funkcí. Otazníky značí neznámou velikost mini-batch

Implementaci chybové funkce **Additive Angular Margin loss (ArcFace)** (rovnice 10) si lze představit jako úpravu Softmax loss. Softmax loss z rovnice 6 lze rozepsat:

$$L = -\log \frac{e^{W_t^T x + b_t}}{\sum_{j=1}^{C} e^{W_j^T x + b_j}}, \tag{26}$$

kde x je vektor příznaků použitý při testování neboli výstup předposlední vrstvy, W jsou váhy poslední vrstvy a b jsou biasy poslední vrstvy. U ArcFace je bias vynechán a váhy s vektory příznaku jsou L2 normalizovány na $||W_j|| = 1$ a $||x|| = 1$. Po L2 normalizaci totiž leží váhy i vektory příznaků na jednotkové nadkouli a při skalárním součinu mezi nimi je výsledkem $\cos\theta_j$. Tohle je základem úhlově založených chybových funkcí jako jsou A-softmax [17], CosFace [18] a samozřejmě ArcFace. Princip za úhlově založenými funkcemi je, že predikce tříd záleží pouze na úhlu θ_j mezi vektorem příznaků x a váhami W_j, kde váhy fungují něco jako centra tříd. Pro zmenšení chyby se tedy všechny váhy sítě optimalizují tak, aby vektory příznaků byly separabilní na nadkouli o poloměru S (hyperparametr funkce), což je velice vhodné, když se při verifikaci obličejů porovnávají tváře pomocí kosinovi podobnosti. Ve funkci ArcFace je navíc penalizační hyperparametr m, který zvětšuje úhel θ_t mezi vektorem příznaků a váhami příslušné třídy W_t, což vede k více diskriminačním příznakům. Po všech těchto úpravách je výsledkem rovnice 10.

Jelikož je nutno manipulovat s váhami, tak stejně jako Center loss, je ArcFace implementována jako vrstva. Váhy jsou inicializovány stejně jako ve výpisu 2, ale jsou nastavené na trénovatelné. Ve výpisu 5 lze vidět výše popsané úpravy. Výstupem vrstvy jsou predikce tříd, takže pro vypočítání chyby je při konfiguraci modelu ještě nutné nastavit `loss=tf.keras.losses.`

`CategoricalCrossentropy()`.

```
def call(self, x, mask=None):
        embeddings, one_hots = x

        norm_embeddings = K.l2_normalize(embeddings, axis=1)
        norm_weights = K.l2_normalize(self.weights, axis=0)
        cos_theta = tf.matmul(norm_embeddings, norm_weights)

        #pridani penalizacni hodnoty m k uhlu theta
        sin_theta = K.sqrt(1 - K.square(cos_theta))
        cos_theta_m = cos_theta * K.cos(self.m) - sin_theta * K.sin(self.m)

        logits = cos_theta * (1 - one_hots) + cos_theta_m * one_hots
        scaled_logits = self.S * logits

        class_predictions = tf.nn.softmax(scaled_logits)

        return class_predictions
```

Výpis 5: Výpočty v poslední vrstvě sítě s chybovou funkcí ArcFace

8.3 Datasety

Počáteční datasety obličejů byly vytvořeny v kontrolovaných prostředích, tzn. osoby byly nafoceny pod podobnými nebo stejnými podmínkami. Takto focené osoby musí souhlasit z účastí a proto je obtížné tímto způsobem vytvořit rozsáhlé datasety. Například známé datasety AT&T a Yale Face Database obsahují pouze 40 a 15 osob. Avšak existují také rozsáhlejší datasety jako FERET, kde 14 126 obrázků bylo sbíráno od roku 1993 až po rok 1996 a obsahují 1199 osob. I když bylo s postupným vývojem oblasti rozpoznání obličejů možné v takovýchto "ideálních" podmínkách dosáhnout vysoké úspěšnosti, tak pro použití modelu v praktické aplikaci byla potřeba generalizace na různé prostředí a také na osoby, které model dosud neviděl. To se sebou přineslo dvě změny - za prvé se vytváření datasetů přeneslo do neomezených (reálných) prostředí a za druhé byli datasety rozděleny na trénovací a testovací. Testovací datasety jsou benchmark datasety, na kterých se natrénované modely porovnávají a dodávají představu jak by se modelu mohlo dařit v praktické aplikaci. Takové datasety mohou být obecné, nebo zaměřené na osoby různých ras, věkových kategoriích a tak dále. Nejznámější obecný testovací dataset v neomezeném prostředí je Labeled Faces in the Wild (LFW), který vyšel v roce 2007. Trénovací datasety pro hluboké učení by měli být co nejvíc rozsáhlé, jak v počtu obrázků (aspoň v řádů

stotisíců), tak počtu osob. Na počátku vývoje hlubokého učení v oblasti rozpoznávání obličejů v roce 2014, měly k takovýmto datasetům přístup pouze velké firmy jako Facebook a Baidu, které si je vytvořily z vlastních zdrojů a jejich zveřejnění není možné kvůli problémům s ochranou soukromí. Pro akademické účely bylo vytvoření takových datasetů se souhlasem focených osob prakticky nemožné a proto se datasety začaly vytvářet z fotek na internetu, na kterých se nachází celebrity a jiné veřejné osobnosti.

Významné datasety pro hluboké učení v rámci rozpoznávání obličejů:

- Již výše zmíněný dataset **Labeled Faces in the Wild (LFW)** [10] je stále standardní benchmark dataset pro verifikaci obličejů, i přesto že vyšel v roce 2007. Dataset obsahuje 13 233 obrázků obličejů od 5749 různých osob. Z toho 4069 osob se vyskytuje z datasetu pouze jednou, což je přesně důvod, proč je dataset určen pro testování a to přesněji pro verifikaci obličejů nežli identifikaci obličejů. Obrázky obličejů se vyskytují v neomezeném prostředí, což znamená velké změny v osvětlení, natočení a výrazů tváří. Dataset byl vytvářen z Yahoo News stránek po dobu zhruba 2 let. Tento dataset je vcelku obecný, ale pár skupin není dostatečně reprezentovaných, jako jsou děti, lidé přes 80 let a několik etnicit. Ohodnocení modelu může spadat do 6 různých protokolů a v rámci hlubokého učení se nejčastěji používá protokol "Unrestricted with labeled outside data", který má jako jediné omezení, že osoby v trénovacích datech se nesmí překrývat s osobami v LFW. Standardně se ohodnocení provádí na předem poskytnutých 6000 párech obrázků obličejů.

- **MegaFace** [38] byl pravděpodobně nejpoužívanější testovací benchmark pro identifikaci obličejů. Skládá se z galerie a probe množiny. Galerie je tvořená MegaFace datasetem, který obsahuje přes 1 milión obrázků a 690 tisíc osob. Probe množina je rozdělena na dataset Facescrub (106 863 obrázků od 530 celebrit) a dataset na otestovaní věkové invariance FGNET (975 obrázků a 82 osob). Ohodnocení modelů v rámci identifikace obličejů se provádí pomocí sestavení pořadí podobnosti mezi testovanými obrázky z probe množiny a všemi obrázky z galerie, která obsahuje alespoň jeden obrázek testované osoby. Distribuce MegaFace datasetu byla bohužel nedávno přerušen, takže již není možné tento testovací benchmark použít.

- **CASIA-Webface** [39] dataset byl vytvořený na konci roku 2014 a je to první veřejný dataset zamýšlený pro trénování modelů hlubokého učení, který začal být široce používaný. Obsahuje 10 575 osob a 494 414 obrázků obličejů celebrit ze stránek IMDb.

- Dalšími rozsáhlejšími veřejnými trénovacimi datasety jsou **MS-Celeb-1M** [40] (zhruba 10 miliónů obrázků obličejů a 100 tisíc osob) a **VGGFace2** [41] (3,31 miliónů obrázků obličejů a 9131 osob) a oba jsou vytvořeny z obrázku celebrit a ostatních veřejných osob z internetu.

- **Google dataset** je největší, ale zato soukromý dataset s 200 milióny obrázků obličejů a 8 miliónu osob, který byl použitý pro natrénování systému FaceNet [13].

8.4 Předzpracování dat

Pro natrénování modelů byl zvolen dataset **CASIA-Webface** [39]. Dataset je oproti jiným novějším datasetům relativně malý, ale pořád je dostatečně velký a kvůli časové náročnosti trénování mnoha modelů v této práci, je CASIA-Webface velice vhodný. Pro otestování natrénovaných modelů byl zvolen dataset **Labeled Faces in the Wild (LFW)** [10]. V obou datasetech mají všechny obrázky velikost 250×250 px. Během předzpracování v kapitolách 8.4.1, 8.4.2 a 8.4.3 jsou vytvářeny nové verze těchto datasetů s velikostí obrázku 96×112 px. Všechny verze jsou shrnuty v tabulkách 5, 6 a dále se na ně budu v textu odkazovat příslušnými jmény.

Označení datasetu	Obrázků	Osob	Poznámky
CASIA-webface	494 414	10 575	Originální dataset.
CASIA-maxpy-clean	455 593	10 575	Podmnožina datasetu CASIA-webface. Pročištěný dataset od špatně označených obrazů.
CASIA-removed	454 103	10 559	Podmnožina datasetu CASIA-maxpy-clean. Dataset bez prolínajících se osob mezi datasety LFW a CASIA.
CASIA-detected	451 938	10 559	Výsledek po detekci obličejů na CASIA-removed.
CASIA-aligned	451 938	10 559	Výsledek po zarovnání obličejů na CASIA-removed.
CASIA-detected-augmented	903 876	10 559	Výsledek po augmentaci dat na CASIA-detected.
CASIA-aligned-augmented	903 876	10 559	Výsledek po augmentaci dat na CASIA-aligned.

Tabulka 5: Přehled všech verzí CASIA datasetu

Označení datasetu	Obrázků	Poznámky
LFW	13 233	Originální dataset.
LFW-detected	13 233	Výsledek po detekci obličejů na LFW.
LFW-aligned	13 233	Výsledek po zarovnání obličejů na LFW-detected.

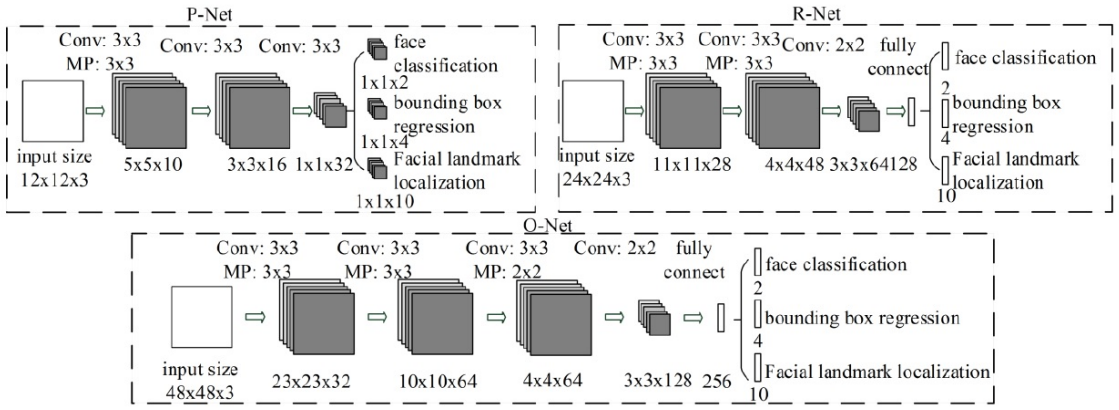
Tabulka 6: Přehled všech verzí LFW datasetu

8.4.1 Detekce tváře

Detekce obličeje je důležitou součástí při rozpoznávání obličejů, jelikož bez detekovaného obličeje není co rozpoznávat. Standardem u trénovacích i testovacích datasetů sice jsou už ořezané obrázky, obvykle s jedním obličejem na každém obrázku, ale i přesto se na nich nachází hodně pozadí. Je tedy dobré provést detekci pro přesnější ořezání obličejů, aby model nemusel pracovat s redundantními informacemi.

Často používaný detektor obličejů se nazývá **Viola-Jones** [35], který byl navržen v roce 2001, jako první robustní a dostatečně rychlý detektor pro detekci v reálném čase. Tento algoritmus je založen na Haarových příznacích a trénování probíhá pomocí AdaBoost algoritmu, kde se vyberou nejdůležitější příznaky a vytvoří se z nich klasifikátory, z kterých jsou nakonec vytvořeny kaskády klasifikátorů. Viola-Jones detekci lze provést například pomocí knihovny OpenCV, která obsahuje již předtrénované kaskády klasifikátorů.

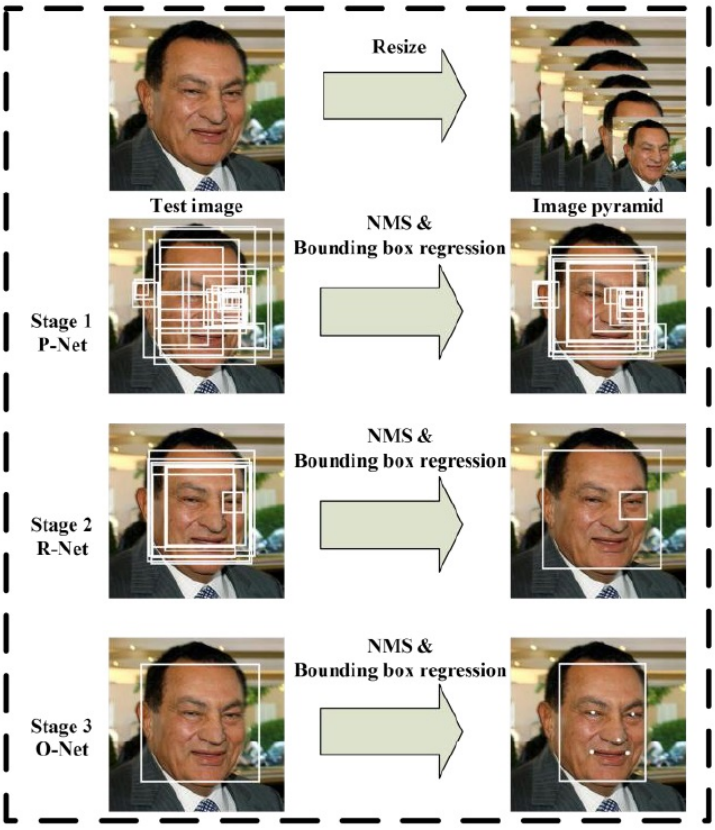
V této práci je použitý novější detektor **Multi-task Cascaded Convolutional Network (MTCNN) neboli víceúčelová kaskáda konvolučních sítí** [36]. Detektor je rozdělený do 3 fází, kde se v každé fázi nachází jedna konvoluční neuronová síť, z čehož taky vychází "kaskáda konvolučních sítí" v názvu detektoru. První část názvu "víceúčelová kaskáda" vychází ze způsobu trénování sítí, které se učí zároveň binární klasifikaci obličeje, regresi ohraničení (bounding box) a lokalizaci landmarků (tzn. regrese souřadnic pěti landmarků). Sítě mají tedy 3 výstupy a každý má svoji chybovou funkci. Trénování sítě probíhá minimalizací váženého součtu všech chyb. Váhy zaleží na dané síti a typu trénovacího obrázku. Například pro negativní obrázek (bez obličeje) se bere v potaz pouze chyba binární klasifikace obličeje. Na obrázku 17 jdou vidět architektury všech 3 sítí. Sítě jsou postupně více komplexní, což taky odpovídá úkolům jednotlivých sítí.



Obrázek 17: Architektury jednotlivých konvolučních sítí v MTCNN [36]

Na obrázku 18 lze vidět, jak funguje detektor během testování. Pro vstupní obrázek jsou nejdříve vytvořeny různé velikosti (image pyramid), aby bylo možné detekovat tváře různých

velikostí. Výstupy posuvného okna, na těchto obrázcích, vstupují do tří fázové kaskády konvolučních sítí. V nejméně složité Proposal Network (P-Net) jsou vytvořeni kandidáti ohraničených obličejů, na které se ještě v rámci první fáze aplikuje non-maximum suppression (NMS), pro spojení příliš překrývajících se kandidátů. Kandidáti poté vstupují do složitější Refinement Network (R-Net), kde jsou negativní kandidáti zamítnuti a pro pozitivní jsou vytvořeny přesnější ohraničení. Na výstup R-Net se také aplikuje NMS a vylepšení kandidáti poté vstupují do nejsložitější Output Network (O-Net), která již vytvoří konečné ohraničení a zároveň landmarky pro oči, nos a koutky úst.



Obrázek 18: MTCNN proces při testování obrázku [36]

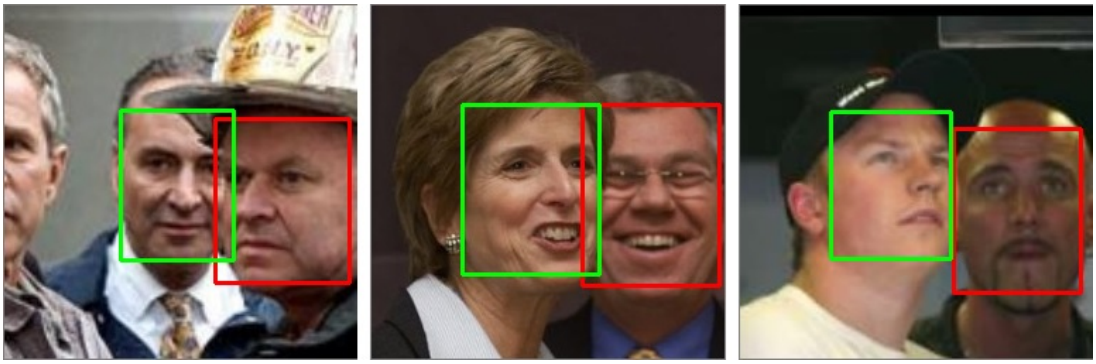
Implementace a předtrénovaný MTCNN model je převzatý z [37]. V této implementaci je navíc možné nastavit hodnotu, podle které je vstupní obrázek postupně zmenšován (image pyramid) a minimální velikost detekovaného obličeje. Ve výpisu 6 jde vidět vytvoření detektoru a jeho výstup při detekci obličeje na obrázku.

```
>>> detector = MTCNN(min_face_size=20, scale_factor=0.709)
>>> detector.detect_faces(img)
[
    {
```

```
        'box': [65, 76, 77, 105],
        'confidence': 0.99851983785629272,
        'keypoints':
        {
            'left_eye': (109, 117),
            'right_eye': (139, 119),
            'nose': (137, 138),
            'mouth_left': (108, 156),
            'mouth_right': (136, 158)
        }
    }
]
```

Výpis 6: Vytvoření a výstup MTCNN detektoru [37]

V CASIA a LFW datasetech patří každý obrázek vždycky jedné osobě. I přesto se ale na obrázcích občas vyskytuje více obličejů, zejména v LFW datasetu. Z toho důvodu jsem ze začátku bral za označenou osobu na obrázku takovou, která má největší detekční okno. Jak jde ale vidět na obrázku 19, tak někdy mají vedlejší osoby vetší detekční okna. Proto jsem se rozhodl počítat centroid pro každé detekční okno a za hlavní osobu na obrázku považuju takovou, která má centroid co nejblíže k centroidu celého obrázku (to mi při testování přidalo přibližně 2% přesnosti). Detekované obličeje jsou oříznuté a zvětšené na 96×112 px a tím jsou vytvořeny datasety CASIA-detected a LFW-detected. V datasetu LFW byly detekovány obličeje na všech obrázcích a v CASIA-removed datasetu na 99,52% ze všech obrázků. Jestli jsou všechny obličeje správně detekovány je těžké určit, ale po delším prohlížení datasetu jsem nenašel žádné vadné detekce. Rychlost zpracování jednoho 250×250 px obrázku s GPU je průměrně 35,6 ms.



Obrázek 19: Obrázky, kde vedlejší osoba (označená červeně) má větší detekční okno než osoba, které tento obrázek patří (označena zeleně).

8.4.2 Zarovnání tváře

Zarovnání tváře je standardní součástí systémů pro rozpoznávání tváří. Správné zarovnání zlepšuje robustnost proti natočení obličeje. V této práci jsou obličeje zarovnány transformační maticí, s kterou je na obrázcích aplikovaná změna měřítka, otočení a posun. Pro vytvoření matice jsou využity landmarky očí a velikost detekčního okna, které jsou získány s detektorem MTCNN. Transformovaný obraz by měl splňovat následující požadavky:

1. obě oči jsou ve stejné úrovni (obličej je otočený tak, aby obě oči měly stejnou souřadnici y)
2. všechny obličeje mají přibližně stejnou velikost
3. střed mezi očima se nachází uprostřed obrazu na ose x
4. úroveň očí je pro všechny obrazy v datasetu stejná

Pro vytvoření transformační matice a její použití je použita knihovna OpenCV. Ve výpisu 7 lze vidět výpočet matice M pomocí funkce *getRotationMatrix2D*, kde *center* značí bod, kolem kterého se provádí rotace, *angle* je úhel otočení a *scale* je faktor změny měřítka. Pro výpočet těchto hodnot je nutno znát landmark levého oka označený jako l, landmark pravého oka označený jako r, výšku výsledného obrazu označenou *height_final* a výšku detekčního okna označenou jako *height_detection*.

```
center = ((l.x + r.x) // 2, (l.y + r.y) // 2)
angle = np.rad2deg(np.arctan2(r.x - l.x, r.y - l.y))
scale = height_final / height_detection
M = cv2.getRotationMatrix2D(center, angle, scale)
```

Výpis 7: Výpočet hodnot pro vytvoření transformační matice

S maticí M je nyní možné provést transformaci, která bude splňovat 1. a 2. požadavek. Pro splnění zbylých požadavků je ještě nutno vypočítat posun. Posun po ose x zajistí splnění 3. požadavku a posun po ose y zajistí splnění 4. požadavku. Výpočet posunu lze vidět ve výpisu 8, kde *eye_level* je úroveň očí, která je nastavená na 0,4.

```
translation.x = width_final * 0.5 - center.x
translation.y = height_final * eye_level - center.y
```

Výpis 8: Výpočet posunu

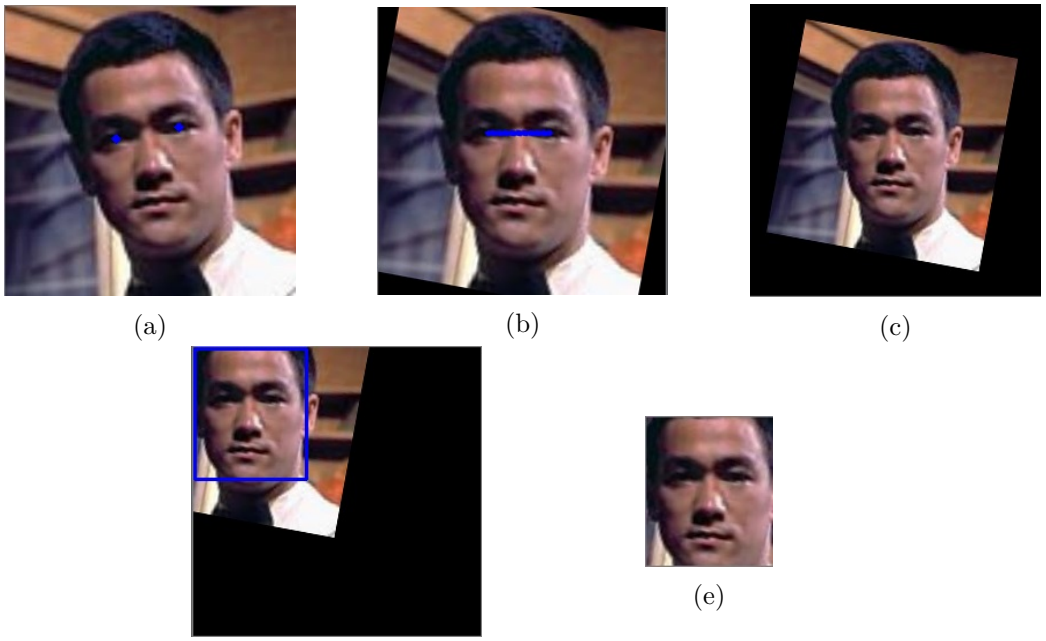
Tento posun je přidán do matice, která následně vypadá:

$$\mathbf{M} = \begin{bmatrix} \alpha & \beta & (1-\alpha)center.x - \beta center.y + translation.x \\ -\beta & \alpha & (1-\alpha)center.y + \beta center.x + translation.y \end{bmatrix}, \tag{27}$$

kde $\alpha = scale \cdot \cos(angle)$ a $\beta = scale \cdot \sin(angle)$. Transformace s touto maticí je provedena pomocí funkce:

*cv2.warpAffine(src, **M**, (width, height), flags = INTER_CUBIC).*

Vizualizace jednotlivých kroků jde vidět na obrázku 21 (i když jsou při transformaci všechny kroky provedeny najednou).



(a) (b) (c) (d) (e)

Obrázek 20: Vizualizace jednotlivých kroků při zarovnávání tváře z obrázku o velikosti 250×250 px na obrázek o velikosti 96×112. **a)** originální obličej s označenými landmarky očí **b)** otočení aby obě oči byly na stejné úrovni **c)** změna měřítka aby se obličej vlezl do výsledného obrazu **d)** posun obličeje (označení naznačuje velikost výsledného obrázku) **e)** výsledný zarovnaný oříznutý obličej

Výsledkem tohoto zarovnání jsou datasety CASIA-aligned a LFW-aligned s obrázky o velikosti 96×112. Jelikož jsou pro zarovnání obličejů nutné landmarky z detektoru MTCNN, tak je v zarovnaných datasetech stejný počet obrázků, jako v detekovaných datasetech (CASIA-detected a LFW-detected).

8.4.3 Augmentace dat

Augmentace dat je proces, ve kterém se zvětšuje rozmanitost a počet dat a tím se předchází přetrénování neuronových sítí. V oblasti rozpoznávání objektů v obrazech to zahrnuje zrcadlení, otáčení, oříznutí v různých místech či změnu úrovně jasu. Jelikož je již v této práci provedeno zarovnání obličejů, tak mnoho těchto operací by byla kontraproduktivní. Proto jsou data augmentovány horizontálním zrcadlením (obrázek 21), což neznehodnotí žádné kroky, které byly provedeny během zarovnání obličejů. Výsledkem augmentace jsou datasety CASIA-detected-augmented a CASIA-aligned-augmented.

Pro zvýšení přesnosti lze také augmentaci aplikovat u testovacích dat. Při testování se tedy vytvoří vektory příznaků pro originální i horizontálně zrcadlený obrázek a ty jsou poté sečteny. U datasetu LFW se nevytváří augmentovaný dataset předem, ale místo toho se pro testované obrázky vytváří jejich horizontální verze za běhu.

(a) originální obrázek (b) po horizontálním zrcadlení

Obrázek 21: Ukázka augmentace na obrázku z datasetu CASIA

8.4.4 Normalizace

Jak již bylo zmíněno v podkapitole 6.1, tak pro normalizaci dat lze použít Min-Max normalizace, její varianta pro normalizaci do rozmezí <-1, 1> a nebo standardizace (z-skóre). Podle ostatních prací v oblasti rozpoznávání tváří [17], [18], [16] jsem zvolil variantu normalizace do rozmezí <-1, 1>.

Normalizace se u obrazů může aplikovat několika způsoby:

- **Na celém datasetu** - nejčastěji používaný způsob normalizace obrazů, při kterém se parametry pro normalizaci vypočítají z celého datasetu. V případě barevných obrazů je lze počítat pro každý RGB kanál zvlášť.
- **Na jednotlivých obrazech** - parametry pro normalizaci se počítají a aplikují na každém obrazu. Hodí se pokud je stejný obraz zachycený jednou v jasnějším osvětlení a podruhé v tmavším a zároveň když tato informace o celkovém jasu není důležitá pro klasifikaci. Ovšem můžeme takhle přijít o relativní změny mezi obrazy.
- **Na jednotlivých "příznacích" (pixelech) přes celý dataset** - parametry pro normalizaci se počítají a aplikují u každého pixelu skrze celý dataset. Dává smysl u tabulových dat, kde mají sloupce různé rozmezí a nemají mezi sebou spojitost, ale u obrazů takhle přijdeme o informace v jednotlivých obrazech.

V této práci je použita normalizace skrze celý dataset a jelikož jsou při práci s velkými datasety zajisté zastoupeny minimální i maximální hodnoty pixelu, tak lze jednoduše provést

$$x' = \frac{x - 127,5}{127,5}. \tag{28}$$

8.5 Průběh testování na datasetu Labeled Faces in the Wild

LFW, jakožto benchmark pro porovnání modelů ve verifikaci obličejů, obsahuje již předem dané páry, na kterých se provádí testování. Celkový počet párů je 6000 a z toho je 3000 pozitivních a 3000 negativních. Páry jsou rozdělené do 10 sad a testování probíhá způsobem 10-násobné křížové validace, kde jedna sada vždy reprezentuje testovací páry a 9 sad reprezentuje validační páry. Validační páry slouží pro nalezení klasifikačního prahu, s kterým je dosažena nejlepší přesnost klasifikace párů. Přesněji mohou při klasifikaci nastat 4 možné výsledky:

- **true positive** (TP) - správně klasifikován pozitivní pár
- **false positive** (FP) - nesprávně klasifikován negativní pár
- **true negative** (TN) - správně klasifikován negativní pár
- **false negative** (FN) - nesprávně klasifikován pozitivní pár

a podle těchto výsledků je vypočítána přesnost:

$$p = \frac{TP + TN}{TP + TN + FP + FN}. \tag{29}$$

Pomocí klasifikačního prahu s největší přesností na validačních párech, je poté vypočítána přesnost na testovacích párech. Celý tento proces je opakován 10-krát, aby se všechna data jednou staly testovacími. Pro platné konečné ohodnocení se očekává předložit průměrnou přesnost (rovnice 30), se střední chybou průměru (rovnice 31).

$$\hat{\mu} = \frac{\sum_{i=1}^{10} p_i}{10} \tag{30}$$

$$S_E = \frac{1}{\sqrt{10}} \cdot \sqrt{\frac{\sum_{i=10}^{10} (p_i - \hat{\mu})^2}{9}} \tag{31}$$

Dále se také často publikuje ROC (Receiver Operating Characteristic) křivka, u které se hodnotí kvalita binárního klasifikátoru na základě změny klasifikačního prahu. ROC křivka popisuje vztah mezi *true positive rate* (rovnice 32) a *false positive rate* (rovnice 33). Pro výpočet ROC křivky se tedy nepoužívá 10-násobná křížová validace, ale rovnou provádí klasifikace na všech 6000 párech, s několika klasifikačními prahy.

$$TPR = \frac{TP}{TP + FN} \tag{32}$$

$$FPR = \frac{FP}{FP + TN} \tag{33}$$

8.6 Trénování a výsledky testování

Veškeré trénování proběhlo na čtyřech grafických kartách NVIDIA GeForce RTX 1070, které mají dohromady velikost operační paměti VRAM 32GB. Takto velká VRAM umožnila natrénování modelů s milióny aktivací, desítek miliónů parametrů a zároveň použití mini-batch o velikosti

v řádu stovek. Jednotlivé mini-batch jsou tedy rozdělovány do 4 stejně velkých části, kde každá část je poslána do jedné z grafických karet. Na grafických kartách jsou vypočítány gradienty a poté pomocí celkového průměru jsou aktualizovány parametry sítě.

Trénování a testování je rozdělené do 3 částí. Nejdříve jsou v podkapitole 8.6.1 provedené experimenty na architekturách z kapitoly 7 a jejich modifikacích. V podkapitole 8.6.2 jsou poté na pár vybraných architekturách provedeny experimenty s různými chybovými funkcemi. Na nejlepší kombinaci architektury a chybové funkce jsou poté provedeny finální experimenty v podkapitole 8.6.3, kde jsou vyzkoušeny různé varianty datasetů, nastavení optimalizátoru a tak dále.

V prvních dvou podkapitolách jsou vytvořené skoro stejné trénovací podmínky, pro platné porovnání mezi jednotlivými modely. Jedinou změnou je hyperparametr learning rate, jelikož jsem si všiml, že jednotlivé typy architektur jsou velice citlivé na počáteční nastavení tohoto hyperparametru. Proto jsem vyzkoušel learning rate=$\{0{,}1;\ 0{,}01;\ 0{,}001\}$ pro všechny typy architektur a s nejlepší hodnotou jsem pokračoval při dalších změnách dané architektury. Nastavení, které jsou stejné pro všechny sítě v podkapitolách 8.6.1 a 8.6.2 jsou:

- Optimalizátor Stochastic gradient descent
- momentum = 0,9
- Počet vzorků (mini-batch) = 256
- L2 regularizace $\lambda = 0,0005$
- Velikost předposlední vrstvy (vrstva ze které je během testování získaný vektor příznaků) = 512
- Xavier inicializace vah (rovnice 22), pokud nezmíněno jinak
- Trénování na datasetu CASIA-aligned-augmented (tzn. všechny vstupní vrstvy mají velikost $96 \times 112 \times 3$ a výstupní vrstvy mají 10559 neuronů). Dataset je rozdělený v poměru 9:1 na trénovací a validační data
- Normalizace obrázků do rozmezí <-1, 1> (rovnice 28)
- Learning rate je dělený 10, pokud se chyba hlavní chybové funkce (s Center loss nebo u Inception sítí je více chybových funkcí) na validačních datech nezmenší alespoň o 0,01. Tohle dělení je v průběhu trénování provedeno maximálně 2x
- Trénování skončí, pokud se chyba hlavní chybové funkce na validačních datech nezmenší v průběhu 3 epoch alespoň o 0,0025 (brzké zastavení, viz kapitola 5.3). Po skončení trénování jsou váhy obnoveny z epochy s nejmenší hodnotou monitorované chybové funkce

Před testováním je pokaždé odstraňena poslední vrstva sítě. Kvalita natrénovaných modelů je v prvních dvou podkapitolách otestovaná na datasetu LFW-aligned. Stejně jako během trénování, jsou obrázky normalizovány do rozsahu <-1, 1>. Na obrázcích je ještě před normalizací provedeno horizontální zrcadlení a vektory příznaků od originálního a zrcadleného obrázku jsou

sečteny. Pro porovnání mezi dvěma obrázky obličejů je použitá kosinova podobnost:

$$podobnost = \frac{\sum_{i=1}^{N} A_i B_i}{\sqrt{\sum_{i=1}^{N} A_i^2}\sqrt{\sum_{i=1}^{N} B_i^2}}, \tag{34}$$

kde N je velikost vektoru příznaků. Pokud je podobnost mezi dvěma vektory vetší než daný klasifikační práh, tak je odhadnuto, že obličeje patří stejné osobě, jinak je odhadnuto, že obličeje patří různým osobám.

8.6.1 Experimenty s architekturami

Ve všech následujících tabulkách je uvedena průměrná přesnost z rovnice 30, střední chyba průměru S_E z rovnice 31, počet epoch a průměrný čas pro dokončení jedné epochy.

Nejdříve jsou provedeny experimenty na sítích s typem architektury AlexNet [46] a výsledky jsou uvedeny v tabulce 7. Hodnota learning rate je nastavena na 0,01. U všech sítí je počet neuronů v předposlední vrstvě nastaven na 512. Kromě téhle změny (a jinak velké vstupní a výstupní vrstvy) je architektura sítě **AlexNet default** stejná, jako v tabulce 2.

Avšak původní AlexNet architektura byla navržena pro obrázky o velikosti 227×227, takže pro zachování přibližně stejné velikosti příznakových map, je krok v první konvoluční vrstvě zmenšený z 4 na 2. Síť s touto změnou je v tabulce pojmenována čistě **AlexNet** a jak je vidět, tak se oproti AlexNet default výrazně zlepšil výkon sítě. Z toho důvodu je tato změna ponechána i v dalších variantách.

AlexNet-plus je obohacena o novější praktiky, které za dob architektury AlexNet nebyly k dispozici. Přesněji je přidaná batch normalizace u každé konvoluční a plně propojené vrstvy a všechny váhy jsou inicializované pomocí He inicializace (rovnice 23). Tato změna znovu zvedla výkon sítě. Jde vidět, že batch normalizace trošku zvedá výpočetní náročnost, ale ne moc.

U **AlexNet-plus w/o ReLU** jsem odebral ReLU z předposlední vrstvy, z které se vytváří vektor příznaků při testování. U této změny jsem doufal, že zvětšení prostoru do záporných hodnot pro vektor příznaků bude prospěšný, ale nestalo se tak.

Dále jsou otestovány tvrzení v práci [25], že batch normalizace umožní použití vyššího learning rate a taky že dodává síti jistou regularizaci. U architektur s **0,1** ve jméně je nastaven learning rate=0,1 a **w/o dropout** znamená odstranění techniky dropout u předposlední vrstvy. Jde vidět, že tyto změny sice zmenšily výkon sítě, ale o hodně méně, než tyto stejné změny u sítí bez batch normalizace. AlexNet 0,1 se učil obzvláště dlouho, oproti AlexNet-plus 0,1, kde se učení ještě zkrátilo.

Architektura	Přesnost [%]	S_E	Epoch	Čas/Epoch [s]
AlexNet default	94,6	0,003	25	365
AlexNet	96,62	0,0026	30	385
AlexNet-plus	**97,2**	0,003	23	405
AlexNet-plus w/o ReLU	97	0,0028	25	407
AlexNet-plus 0,1	96,93	0,0032	19	408
AlexNet 0,1	94,55	0,0029	57	384
AlexNet-plus w/o dropout	96,98	0,0018	25	407
AlexNet w/o dropout	94,88	0,0028	17	385

Tabulka 7: Výsledky testování s natrénovanými sítěmi typu AlexNet

Další experimenty jsou provedeny na VGG sítích [50] a výsledky jsou uvedeny v tabulce 8. Hodnota learning rate je nastavena na 0,01. Architektura **VGG16 default** je zobrazena v příloze A a je to klasická VGG16 architektura, se stejnými změnami jako u sítě AlexNet default, tzn. jiná vstupní, výstupní vrstva a předposlední vrstva je zmenšena na 512 neuronů.

Jelikož VGG architektury byly stejně jako AlexNet navrženy pro větší vstupní obrázky, tak pro zachování stejného počtu propojení mezi poslední konvoluční vrstvou a první plně propojenou vrstvou, je odstraněna poslední max pool vrstva (v tabulce pod jménem **VGG16**). Tato změna lehce zvětšila přesnost a proto je aplikovaná u všech ostatních variant. Zkoušel jsem také tuto změnu provést na začátku sítě, aby byly stejně velké i příznakové mapy, ale jednotlivé epochy trvaly příliš dlouho.

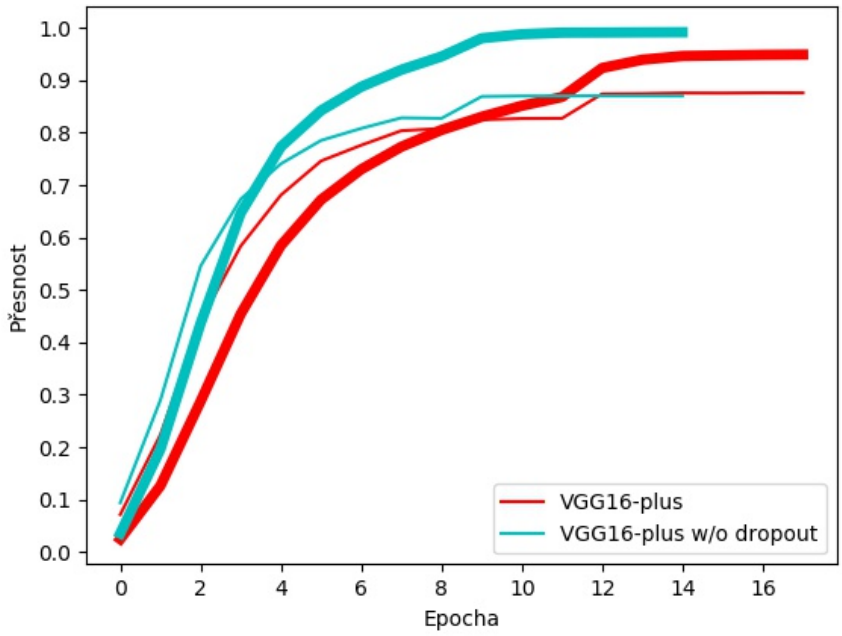
Všechny ostatní označení nesou stejný význam jako v tabulce 7. Přidání batch normalizace v **VGG16-plus** výrazně zvedlo přesnost, stejně jako tomu bylo u AlexNet-plus. Mezi VGG16 a **VGG16 w/o dropout** je překvapivě menší mezera v přesnosti, než mezi VGG16-plus a **VGG16-plus w/o dropout**.

Na obrázku 22 je vykreslena přesnost během trénování u sítí VGG16-plus a VGG16-plus w/o dropout. Jde vidět, jak se síť bez techniky dropout dokáže mnohem lépe naučit trénovací data, ale zato se jí hůře daří na validačních datech (a také testovacích datech z LFW).

Nejlépe se dařilo síti VGG16-plus, tak jsem ještě zkusil stejné změny aplikovat na hlubší verzi s 19 vrstvami (struktura architektury je v příloze B). Tato verze pojmenována **VGG19-plus** měla podobnou, ale stále menší přesnost.

Architektura	Přesnost [%]	S_E	Epoch	Čas/Epoch [s]
VGG16 default	96.68	0,0028	22	781
VGG16	96,82	0,0032	21	1008
VGG16-plus	**98,22**	0,002	18	1104
VGG16-plus 0,1	97,87	0,0031	17	1105
VGG16 0,1	95,13	0,0019	22	1014
VGG16 plus w/o dropout	97,53	0,0029	15	1097
VGG16 w/o dropout	96,25	0,003	19	1006
VGG19-plus	98,05	0,0029	19	1235

Tabulka 8: Výsledky testování s natrénovanými sítěmi typu VGG



Obrázek 22: Porovnání trénování VGG16-plus a VGG16-plus w/o dropout. Tlustá čára značí přesnost na trénovacích datech a tenká čára značí přesnost na validačních datech

V tabulce 9 jsou výsledky experimentů na Inception sítích [47]. Hodnota learning rate je nastavena na 0,01. Předloha pro použité Inception sítě je v tabulce 3, kde jsou ještě napojené pomocné klasifikátory (příloha C) na výstup inception modulů 4a a 4d. Chyba obou pomocných klasifikátorů je přičtena k celkové chybě sítě s váhami 0,3. Během testování jsou pomocné klasifikátory ignorovány.

Nejdříve je pro vytvoření sítě **Inception-v1 default** zmenšená velikost předposlední vrstvy na 512 neuronů. Avšak oproti AlexNet a VGG sítím, je zde předposlední vrstva global average

pooling. To znamená, že pro nastavení velikosti vektoru příznaku nelze přímo nastavit počet neuronů jako u plně propojené vrstvy, ale je nutné změnit počet výstupních příznakových map z posledního inception modulu. Výchozí počet příznakových map je 1024, takže je potřeba tento počet snížit na polovinu. To jsem uskutečnil snížením počtu filtrů na polovinu, u všech výstupních vrstev v daném inception modulu.

V síti **Inception-v1** je zmenšený krok první konvoluční vrstvy z 2 na 1, pro zachování stejného počtu příznakových map, jako v originálním GoogLeNet. Tato změna, stejně jako u AlexNet, výrazně zlepšila přesnost a je využita i u dalších Inception variant.

U **Inception-v1 w/o dropout** je odstraněn dropout na konci sítě (u pomocných klasifikátoru je ponechán) a přesnost výrazně klesla, takže i u sítí bez velkého počtu parametrů je dropout prospěšný.

V sítí **Inception-v1 FC** je global average pool vrstva nahrazena za plně propojenou vrstvu s 512 neurony. To umožnilo obnovit poslední inception modul do původní podoby, ale přesnost stejně klesla.

V sítí **Inception-v1 w/o aux** jsou odebrány pomocné klasifikátory, které podle [31] přidávají regularizaci. Jak se dalo očekávat, tak přesnost zde oproti Inception-v1 znovu klesá.

Inception-v2 síť [25], někdy také označována Inception-BN, je v podstatě síť Inception-v1 s přidanou batch normalizací a He inicializací vah. Přidání batch normalizace znovu zvedla přesnost, i když ne až o tolik, jako u typu architektury AlexNet a VGG. Po odebrání pomocných klasifikátorů v **Inception-v2 w/o aux** není zaznamenána moc velká změna, takže by šlo tvrdit, že batch normalizace pomáhá s regularizací sítě.

Architektura	Přesnost [%]	S_E	Epoch	Čas/Epoch [s]
Inception-v1 default	95,8	0,0033	23	384
Inception-v1	97,08	0,0029	48	703
Inception-v1 w/o dropout	94,85	0,0036	21	703
Inception-v1 FC	96,43	0,0022	23	742
Inception-v1 w/o aux	96,17	0,0036	35	642
Inception-v2	**97,35**	0,0027	23	800
Inception-v2 w/o aux	97,32	0,0024	21	740

Tabulka 9: Výsledky testování s natrénovanými sítěmi typu Inception

Poslední sítě na otestování jsou ResNet [51] a výsledky těchto testů jdou vidět v tabulce 10. Hodnota learning rate je nastavena na 0,1. U všech sítí je použita He inicializace vah. V příloze E je zobrazena struktura architektury pro síť **ResNet34 default**. Tato architektura je stejná jako v originální práci [51] a jelikož je již výchozí velikost předposlední vrstvy 512, tak nejsou nutné žádné změny. Pokud se v reziduálním bloku provádí zmenšení příznakových map, tak jsou ve zkratce použity 1×1 konvoluce, aby seděly dimenze vstupu a výstupu (kvůli sčítaní příznakových map na konci bloku).

V síti **ResNet34** je odebraná první max pool vrstva, pro zachování stejně velkých příznakových map, jako v originální ResNet architektuře. Tato změna výrazně zlepšila přesnost a je použita u všech dalších ResNet variant. Také jsem zkoušel zmenšit krok z 2 na 1 v první konvoluční vrstvě (jak jsem dělal u sítí Inception), ale odebrání max pool vrstvy vedlo k trošku lepší přesnosti.

Dále je vyzkoušena architektura ResNet50. Avšak výchozí velikost předposlední vrstvy v této architektuře je 2048. Proto jsem vyzkoušel několik zakončení této architektury, ale jediná se slušnou přesností je **ResNet50-BN-dropout-FC-BN** (podrobněji zobrazena v příloze E). I přesto má ResNet34 lepší přesnost, tak jsem se rozhodl nezkoušet hlubší varianty.

Pre-ResNet34 je Pre-activation ResNet34 [33], kde je přemístěná aktivační funkce ReLU před sčítání příznakových map a pořadí operací uvnitř reziduálních bloků je změněno na BN (Batch normalizace)->ReLU->Konvoluce->BN->ReLU->Konvoluce. Tato varianta dosáhla větší přesnosti než klasická ResNet34.

Jelikož se v Pre-activation reziduálním bloku mění pořadí operací, tak do global average pool vrstvy, ze které je brán vektor příznaku, vstupuje výstup přímo z konvoluční vrstvy (bez aktivační funkce). Oproti tomu, v ResNet34 vstupuje do global average pool vrstvy výstup z funkce ReLU. Proto jsem v **Pre-ResNet34 extra ReLU** zkusil dát před global average pool vrstvu funkci ReLU, ale přesnost s touto změnou klesla.

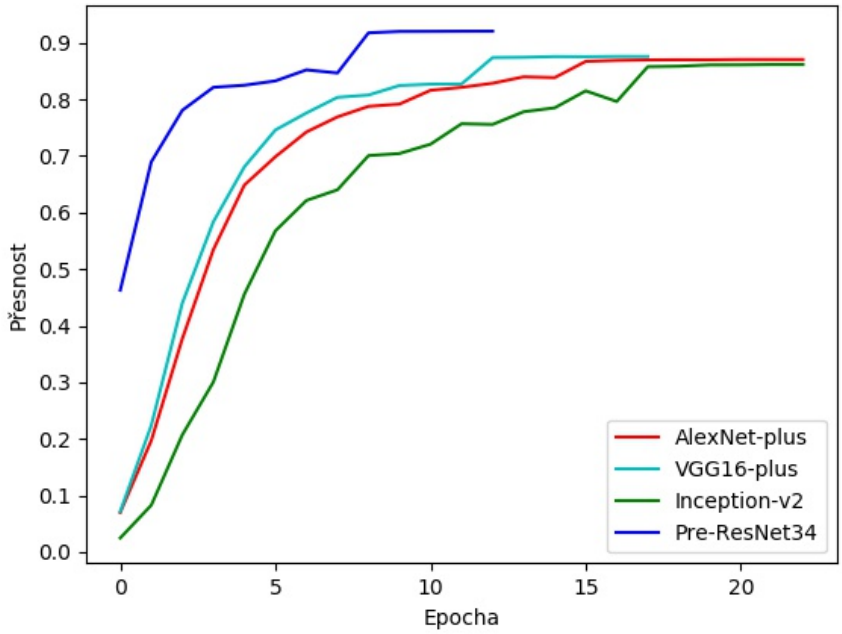
Architektura	Přesnost [%]	S_E	Epoch	Čas/Epoch [s]
ResNet34 default	95,9	0,003	19	435
ResNet34	97,02	0,0032	20	983
ResNet50-BN-dropout-FC-BN	96,38	0,003	18	1975
Pre-ResNet34	**97,17**	0,0022	13	915
Pre-ResNet34 extra ReLU	96,78	0,0027	14	913

Tabulka 10: Výsledky testování s natrénovanými sítěmi typu ResNet

Nejlepší sítě z každého typu architektury jsou shrnuty v tabulce 11. Batch normalizace se ukázala být důležitou a zároveň časově nenáročnou součástí. Podstatně jasný náskok má síť VGG16-plus, jinak jsou na tom ostatní sítě podobně. Překvapivě měl zástupce ResNet sítí nejmenší přesnost na LFW datasetu, i když dosáhl během trénování největší validační přesnosti (obrázek 23).

Architektura	Přesnost [%]	S_E	Epoch	Čas/Epoch [s]
Pre-ResNet34	97,17	0,0022	13	915
AlexNet-plus	97,2	0,003	23	405
Inception-v2	97,35	0,0027	23	800
VGG16-plus	98,22	0,002	18	1104

Tabulka 11: Nejlepší sítě z každého typu architektury



Obrázek 23: Porovnání přesností na validačních datech u sítí z tabulky 11

8.6.2 Experimenty s chybovými funkcemi

Na sítích z tabulky 11 jsou v této podkapitole vyzkoušeny chybové funkce Center loss [15] a ArcFace [16].

Trénování sítě s **Center loss** je doprovázeno s funkcí Softmax loss, jak je popsáno v kapitole 8.2 a během testování je vrstva s Center loss odstraněna. Výsledky testování jsou zobrazeny v tabulce 12, kde hyperparametry Center loss jsou podle originální práce [15] nastaveny na $\lambda = 0,003$ a $\alpha = 0,5$. Na síti s největší přesností Pre-Resnet34 jsem vyzkoušel i $\lambda = 0,0005$ a $\lambda = 0,01$, ale původní nastavení fungovalo nejlépe. Změny v přesnosti nebyly tak výrazné, tak jsem se rozhodl u ostatních sítí jiné λ hodnoty nezkoušet.

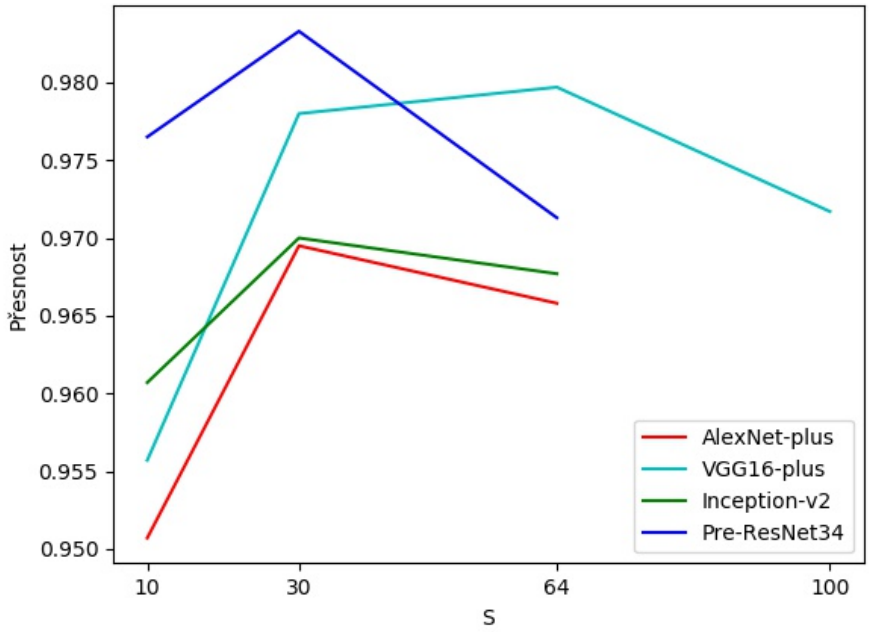
Sítě s technikou dropout v předposlední vrstvě se učily pomalu, tak jsem zkusil ji u zakončení sítě s Center loss odstranit (u zakončení se Softmax loss dropout zůstal). Sítě s touto změnou jsou v tabulce označeny **before dropout** a jde vidět, že se nejenom zrychlilo trénovaní sítě, ale taky se zvětšila přesnost. U VGG16-plus bylo trénování tak pomalé, že jsem natrénoval pouze variantu VGG16-plus before dropout.

V porovnání se sítěmi s Softmax loss (tabulka 11), se přesnost zlepšila u sítí Inception-v2 a Pre-ResNet34 a doba pro dokončení epochy nepatrně vzrostla.

Architektura	Přesnost [%]	S_E	Epoch	Čas/Epoch [s]
AlexNet-plus	96,52	0,0018	39	414
AlexNet-plus before dropout	97,05	0,0023	23	422
VGG16-plus before dropout	97,52	0,003	21	1120
Inception-v2	97,53	0,0025	28	910
Inception-v2 before dropout	97,67	0,0027	20	915
Pre-ResNet34	**97,75**	0,0021	11	927
Pre-ResNet34 $\lambda = 0,01$	97,68	0,0028	12	927
Pre-ResNet34 $\lambda = 0,0005$	97,7	0,0024	14	927

Tabulka 12: Výsledky testování sítí s chybovou funkcí Center loss

U chybové funkce **ArcFace** jsem nejprve experimentoval s hyperparametrem S (obrázek 24), zatímco penalizační hyperparameter m byl nastaven na 0. Pro každou síť takhle byla vybrána nejlepší hodnota S a v tabulkách 13 a 14 jsou zobrazeny experimenty s hyperparametrem m. U sítí s technikou dropout v předposlední vrstvě má hyperparametr m větší sílu a proto jsou jenom u Pre-ResNet34 vybrané větší hodnoty m. Na obrázku 25 jsou zobrazeny přesnosti na konci trénování, podle změny hyperparametru m, pro sítě VGG16-plus a Pre-ResNet34. Jde vidět, že i když je rozsah hyperparametru u obou sítí o hodně jiný, tak mají na trénovacích datech podobný efekt.



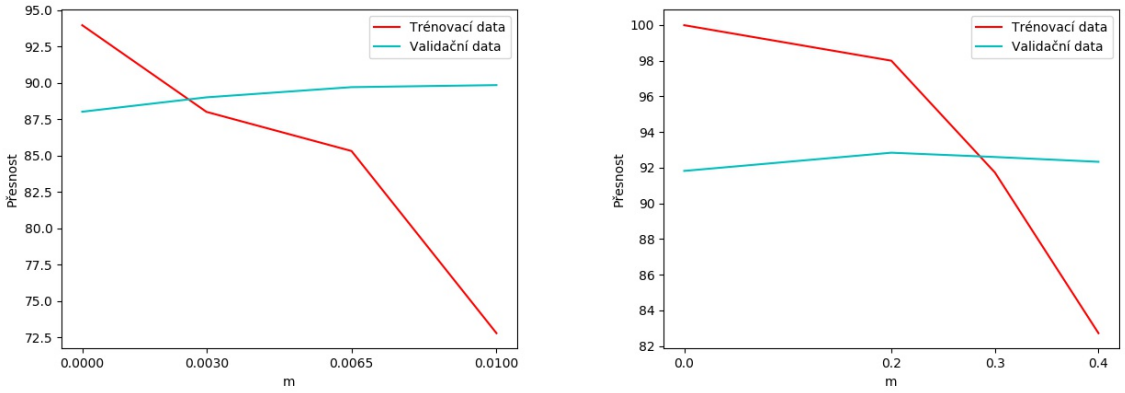
Obrázek 24: Výsledky testování se změnou hyperparametru S u chybové funkce ArcFace

Architektura	S	Přesnost [%]				Čas/Epoch [s]
		m=0	m=0,03	m=0,065	m=0,1	
AlexNet-plus	30	96,5	97,08	**97,25**	97,18	420
Inception-v2	30	97,25	97,82	**98**	97,95	823
VGG16-plus	64	97,97	**98,13**	98,07	98,12	1115

Tabulka 13: Výsledky testování s chybovou funkcí ArcFace na sítích s technikou dropout v předposlední vrstvě

Architektura	S	Přesnost [%]				Čas/Epoch [s]
		m=0	m=0,2	m=0,3	m=0,4	
Pre-ResNet34	30	98,33	98,72	**98,87**	98,63	920

Tabulka 14: Výsledky testování s chybovou funkcí ArcFace na jediné síti bez techniky dropout v předposlední vrstvě



(a) VGG16-plus

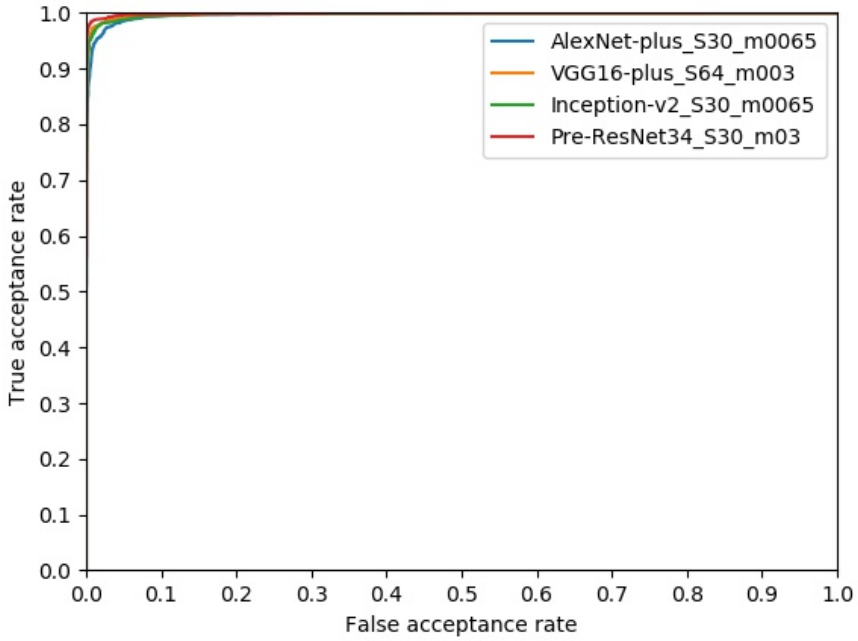
(b) Pre-ResNet34

Obrázek 25: Porovnání síly efektu penalizačního hyperparametru m. Během trénování je hyperparametr aplikován, takže jde vidět jeho efekt, ale během validace není aplikován, aby šlo vidět jak se síti doopravdy daří

V tabulce 15 jsou shrnuty nejlepší přesnosti pro každou chybovou funkci a každý typ architektury. Na obrázku 26 je zobrazeny ROC křívky pro sítě s největší přeností u každého typu architektury. I když Center loss vylepšila 2 sítě oproti Softmax loss, tak funkce ArcFace jí vždycky překonala. Síť Pre-ResNet34 měla nejmenší přesnost s funkcí Softmax loss, ale s funkcí ArcFace má celkově nejvyšší přesnost, dokonce i s výrazným náskokem.

Architektura	Přesnost [%]		
	Softmax loss	Center loss	ArcFace
AlexNet-plus	97,2	97,05	**97,25**
Inception-v2	97,35	97,67	**98**
VGG16-plus	**98,22**	97,52	98,13
Pre-ResNet34	97,17	97,75	**98,87**

Tabulka 15: Shrnutí výsledků pro každou chybovou funkci a každý typ architektury



Obrázek 26: ROC křivka pro nejlepší sítě z každého typu architektury

8.6.3 Finální experimenty

V této podkapitole jsou provedeny finální experimenty na dosuď nejlepším modelu, kterým je Pre-ResNet34 s chybovou funkcí ArcFace ($S = 30$, $m = 0,3$).

V tabulkách 16, 17 a 18 je otestováno, jestli měly jednotlivé kroky předzpracování v kapitole 8.4 význam. Tabulky jsou vytvořeny tak, aby šel v jednotlivých krocích vidět výsledek každé kombinace předzpracování na trénovacích a testovacích obrázcích. Označení *mirror* u LFW datasetů znamená, že při testování je provedeno horizontální zrcadlení a vektory příznaků od originálního a zrcadleného obrázku jsou sečteny.

Dataset	LFW-aligned mirror	LFW-detected mirror
CASIA-aligned-augmented	98,87	97,17
CASIA-detected-augmented	98,02	98,55

Tabulka 16: Výsledky testování bez a se zarovnáním obličeje na trénovacím a testovacím datasetu

Dataset	LFW-aligned mirror	LFW-aligned
CASIA-aligned-augmented	98,87	98,63
CASIA-aligned	98,47	98,23

Tabulka 17: Výsledky testování bez a s horizontálním zrcadlením na trénovacích a testovacích obrázcích

	Normalizace při testování	Bez normalizace při testování
Normalizace při trénování	98,87	65,98
Bez normalizace při trénování	60,87	98,3

Tabulka 18: Výsledky testování bez a s normalizací dat na trénovacím datasetu CASIA-aligned-augmented a testovacím datasetu LFW-aligned mirror

V tabulce 16 lze vidět, že je nejlepší zarovnávat obličeje na trénovacích i testovacích obrázcích. Nevyplatí se však zarovnávat obličeje pouze na trénovacích, nebo na testovacích obrázcích. Horizontální zrcadlení je prospěšné, i když je provedeno jenom na trénovacích, nebo testovacích obrázcích, ale nejlepší je provést na obou (tabulka 17). Pokud síť dostává během testování jiné rozdělení dat než na jakém byla natrénována, tak se síti daří velice špatně (tabulka 18). Největší přesnost má síť s normalizovanými vstupními obrázky.

V tabulkách 19 a 20 je zobrazen pokus o doladění hyperparametrů SGD, ale počáteční hodnoty dosáhli nejlepších výsledků. Při odstranění momentum, nebo L2 regularizace se taktéž trochu zhorší přesnost klasifikace (tabulka 21).

Learning rate	Přesnost [%]	S_E	Epoch
0,075	98,78	0,0016	15
0,1	98,87	0,0015	11
0,125	98,55	0,0014	13

Tabulka 19: Výsledky testů při změně hodnoty learning rate

Mini-batch velikost	Přesnost [%]	S_E	Epoch	Čas/Epoch [s]
128	98,62	0,0012	13	1121
256	98,87	0,0014	11	920
512	98,62	0,0014	14	845

Tabulka 20: Výsledky testů při změně velikosti mini-batch

Nastavení	Přesnost [%]	S_E	Epoch
Bez momentum	98,58	0,0016	11
Bez L2 regularizace	98,52	0,0016	13

Tabulka 21: Výsledky testování při odstranění momentum a L2 regularizace

Na datasetu Labeled Face in the Wild byla nakonec dosažena největší přesnost 98,87% se sítí **Pre-ResNet34**, když byla použita chybová funkce **ArcFace** ($S = 30$**,** $m = 0,3$) a když byly provedeny všechny kroky předzpracování z kapitoly 8.4. Hyperparametry SGD již byly překvapivě optimálně nastaveny, když byly experimentálně vybrány na začátku kapitoly 8.6. Celkový počet špatných klasifikací je 68 z 6000 a z toho je 44 FP a 24 FN. Při prozkoumání FN neboli špatně klasifikovaných pozitivních párů jsem si všiml, že mnoho z nich mají buďto hodně jiný výraz, nebo před obličeji překáží nějaký objekt. U těchto párů také dochází k poměrně velké změně osvětlení a natočení obličeje, ale to se děje skoro u všech testovacích párů. U FP neboli špatně klasifikovaných negativních párů musím uznat, že některé páry vypadají dosti podobně. Příklady špatných klasifikaci jsou na obrázku 27.

V LFW je 7 známých špatně označených párů. I když se o těchto špatně označených párech ví, tak kvůli tak malému počtu jsou v datasetu ponechány, aby novější výsledky neměli výhodu nad staršími. Po zkontrolování špatně klasifikovaných párů jsem zjistil, že 6 z nich patří do těchto špatně označených. To znamená, že 6 ze špatných klasifikací je správných a 1 správná klasifikace je špatná. Když tohle vezmu v úvahu, tak to sice není velké navýšení přesnosti, ale síť by měla mít správně přesnost 98,95%.

(a) FN (b) FP

Obrázek 27: Pár příkladů špatně klasifikovaných párů z datasetu LFW-aligned.

9 Závěr

Cílem této práce bylo provést rešerši metod v oblasti rozpoznávání obličejů v obrazech, vytvořit rozpoznávač tváří pomocí konvolučních neuronových sítí a ten poté vyzkoušet na vhodném datasetu. Jelikož již rešerše starších metod byla náplní mojí bakalářské práce, tak po krátkém úvodu do problematiky rozpoznávání obličejů jsem rovnou přešel na nejmodernější metodu v této oblasti - konvoluční neuronové sítě. Při rešerši jsem se zaměřil na metody a techniky, které jsou používány ve state-of-the-art neuronových sítích a také architektury, které přináší úplně nové nápady jak sestavit KNS. Všechny tyto informace mi v praktické části umožnily natrénovat modely, které jsou schopné rozpoznávání obličejů v reálném prostředí.

Práce byla realizována v programovacím jazyce Python s knihovnou Tensorflow, především s balíčkem tf.keras. Při popisu implementace jsem se hlavně věnoval chybovým funkcím, jelikož jejich implementace není součástí knihovny a také protože v oblasti rozpoznávání obličejů byla na vývoj chybových funkcí v posledních letech zaměřena velká pozornost. Pro trénování modelů byl zvolen dataset CASIA-Webface s 10 575 osobami a 494 414 obrázky obličejů celebrit. Pro testování byl zvolen známý benchmark dataset Labeled Faces in the Wild, kde testování probíhá formou verifikace obličejů na 6000 párech. Na těchto vybraných datasetech byly provedeny standartní kroky předzpracovaní u rozpoznávání obličejů a také bylo prokázáno, že všechny kroky byly užitečné, jak na trénovacím, tak i na testovacím datasetu. Při experimentech jsem nejdříve zkoušel architektury AlexNet, VGG, Inception a ResNet. Při menších modifikacích základních variant těchto architektur se mi podařilo docela dost zvednout přesnost na LFW. Při experimentech s chybovými funkcemi byla nejúspěšnější funkce ArcFace. Po pečlivém trénování a testování modelů se mi nakonec podařilo dosáhnout přesnosti 98,87% na LFW se sítí Pre-ResNet34, ve které byla použita chybová funkce ArcFace. Možné vylepšení by mohlo být natrénovat síť na větším datasetu. Například veřejný dataset MS-Celeb-1M [40] obsahuje více než 20x více obrázků, v porovnání s CASIA-Webface, avšak trénovaní by bylo příliš časově náročné.

Literatura

[1] O. Matan et al. *Reading handwritten digits: a ZIP code recognition system*, Computer, vol. 25, no. 7, pp. 59-63, 1992

[2] Matthew D. Zeiler, Rob Fergus. *Visualizing and Understanding Convolutional Networks*, European Conference on Computer Vision, 2013

[3] M. A. Turk and A. P. Pentland. *Face recognition using eigenfaces*. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition(CVPR'91), pp. 586–591, 1991

[4] P. N. Belhumeur, J. P. Hespanha, and D. J. Kriegman. *Eigenfaces vs. fisherfaces: recognition using class specific linear projection*. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 19, no. 7, pp. 711–720, 1997

[5] T. Ahonen, A. Hadid, and M. Pietikainen. *Face description with local binary patterns: Application to face recognition*. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 28, no. 12, pp. 2037–2041, 2006

[6] David G. Lowe. *Distinctive Image Features from Scale-Invariant Keypoints*, International Journal of Computer Vision 60, pp. 91–110, 2004

[7] N. Dalal and B. Triggs. *Histograms of oriented gradients for human detection*. 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), vol. 1, pp. 886–893, 2005

[8] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. *ImageNet Large Scale Visual Recognition Challenge*. International Journal of Computer Vision (IJCV), vol. 115, no. 3, pp. 211–252, 2015

[9] Y. Taigman, M. Yang, M. Ranzato, L. Wolf. *DeepFace: Closing the Gap to Human-Level Performance in Face Verification*, IEEE Conference on Computer Vision and Pattern Recognition, pp. 1701-1708, 2014

[10] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments*, University of Massachusetts, Amherst, Technical Report 07-49, 2007

[11] R. Hadsell, S. Chopra and Y. LeCun. *Dimensionality Reduction by Learning an Invariant Mapping*, IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), pp. 1735-1742, 2006

[12] Y. Sun, X. Wang, X. Tang. *Deep Learning Face Representation from Predicting 10,000 Classes*, IEEE Conference on Computer Vision and Pattern Recognition, pp. 1891-1898, 2014

[13] Florian Schroff, Dmitry Kalenichenko, James philbin. *FaceNet: A unified embedding for face recognition and clustering*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015

[14] Omkar M. Parkhi, Andrea Vedaldi, Andrew Zisserman. *Deep Face Recognition*, Proceedings of the British Machine Vision Conference (BMVC), pp. 41.1-41.12, 2015

[15] Yandong Wen, Kaipeng Zhang, Zhifeng Li, Yu Qiao. *A Discriminative Feature Learning Approach for Deep Face Recognition*, Computer Vision - ECCV 2016, pp. 499-515, 2016

[16] Jiankang Deng, Jia Guo, Niannan Xue, Stefanos Zafeiriou. *ArcFace: Additive Angular Margin Loss for Deep Face Recognition*, IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4685-4694, 2019

[17] Weiyang Liu, Yandong Wen, Zhiding Yu, Ming Li, Bhiksha Raj, Le Song. *SphereFace: Deep Hypersphere Embedding for Face Recognition*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 6738-6746, 2017

[18] Hao Wang, Yitong Wang, Zheng Zhou, Xing Ji, Dihong Gong, Jingchao Zhou, Zhifeng Li, Wei Liu. *CosFace: Large Margin Cosine Loss for Deep Face Recognition*, IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2018

[19] Kunihiko Fukushima. *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position*, Biol. Cybernetics 36, pp. 193–202, 1980

[20] LeCun Yann, Bernhard E. Boser, John S. Denker, Donnie Henderson, R. E. Howard, Wayne E. Hubbard, Lawrence D. Jacke. *Handwritten Digit Recognition with a Back-Propagation Network*, Advances in Neural Information Processing Systems 2, pp. 396-404, 1990

[21] Diederik P. Kingma, Jimmy Ba. *Adam: A Method for Stochastic Optimization*, Proceedings of the 3rd International Conference on Learning Representations (ICLR), 2014

[22] Xavier Glorot, Yoshua Bengio. *Understanding the difficulty of training deep feedforward neural networks*, Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR 9, pp. 249-256, 2010

[23] Xavier Glorot, Antoine Bordes, Yoshua Bengio. *Deep Sparse Rectifier Neural Networks*, Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, pp. 315-323, 2011

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), pp. 1026-1034, 2015

[25] Sergey Ioffe, Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, pp. 448-456, 2015

[26] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, Aleksander Madry. *How Does Batch Normalization Help Optimization?*, 2018

[27] Jeremy Jordan. *Normalizing your data (specifically, input and batch normalization)* [online], 24. leden 2018 [cit. 5. Května 2020]. Dostupné z: https://www.jeremyjordan.me/batch-normalization/

[28] LeCun Y., Bottou L., Orr G.B., Müller K.R. *Efficient BackProp*, Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, vol 1524, 1998

[29] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*,Journal of Machine Learning Research 15(1), pp. 1929-1958, 2014

[30] Dewang Nautiyal. *Underfitting and Overfitting in Machine Learning* [online], [cit. 5. Května 2020], Dostupné z: https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/

[31] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna. *Rethinking the Inception Architecture for Computer Vision*, Computer Vision and Pattern Recognition, 2016

[32] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, Alex Alemi. *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*, AAAI'17: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, pp. 4278-4284, 2016

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Identity Mappings in Deep Residual Networks*, European Conference on Computer Vision, 2016

[34] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, Kaiming He. *Aggregated Residual Transformations for Deep Neural Networks*, Conference: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016

[35] P. Viola and M. Jones. *Rapid object detection using a boosted cascade of simple features*. IEEE Computer Society Conference on Computer Vision and Pattern Recognition(CVPR'01), vol. 1, pp. 511-518, 2001

[36] Zhang Kaipeng, Zhang Zhanpeng, Li Zhifeng, Qiao Yu. *Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks*, IEEE Signal Processing Letters, vol. 23, no. 10, pp. 1499-1503, 2016

[37] Iván de Paz Centeno. GitHub repository, Dostupné z: https://github.com/ipazc/mtcnn

[38] Ira Kemelmacher-Shlizerman, Steve Seitz, Daniel Miller, Evan Brossard. *The MegaFace Benchmark: 1 Million Faces for Recognition at Scale*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016

[39] Dong Yi, Zhen Lei, Shengcai Liao, Stan Z. Li. *Learning Face Representation from Scratch*, 2014

[40] Yandong Guo, Lei Zhang, Yuxiao Hu, Jianfeng Gao. *MS-Celeb-1M: A Dataset and Benchmark for Large-Scale Face Recognition*, European Conference on Computer Vision, 2016

[41] Qiong Cao, Li Shen, Weidi Xie, Omkar M. Parkhi, Andrew Zisserman. *VGGFace2: A dataset for recognising faces across pose and age*, 13th IEEE International Conference on Automatic Face & Gesture Recognition, 2018

[42] Matin Abadi et al. *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015, Dostupné z: tensorflow.org

[43] Bradski G. *The OpenCV Library*, Dr. Dobb's Journal of Software Tools, 2000

[44] Pedregosa et al. *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research, vol. 12, pp. 2825-2830, 2011

[45] Yann Lecun, Leon Bottou, Yoshua Bengio, Patrick Haffner. *Gradient-Based Learning Applied to Document Recognition*, Proceedings of the IEEE, vol. 86, pp. 2278-2324, 1998

[46] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*, Advances in neural information processing systems 25, 2012

[47] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelo, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. *Going deeper with convolutions*, The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015

[48] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-Fe. *ImageNet Large Scale Visual Recognition Challenge*, International Journal of Computer Vision, vol. 115, pp. 211-252, 2014

[49] Jie Hu, Li Shen, Gang Sun. *Squeeze-and-Excitation Networks*, 2017

[50] Karen Simonyan, Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Deep Residual Learning for Image Recognition*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770-778, 2016

[52] Min Lin, Qiang Chen, Shuicheng Yan. *Network In Network*, 2013

[53] Daphne Cornelisse. *An intuitive guide to Convolutional Neural Networks* [online], 24. Duben 2018 [cit. 5. Května 2020]. Dostupné z: https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050

[54] Davis E. King. *Dlib-ml: A Machine Learning Toolkit*, Journal of Machine Learning Research, vol. 10, pp. 1755-1758, 2009

[55] François Chollet et al. *Keras*, 2015. Dostupné z: https://keras.io

A VGG16 architektura

Vrstva	**Počet filtrů**	**Velikost filtrů**	**Velikost výstupu**	**Počet parametrů**
Vstupní			$112 \times 96 \times 3$	
Konvoluční 1.	64	3×3	$112 \times 96 \times 64$	1792
Konvoluční 2.	64	3×3	$112 \times 96 \times 64$	36 928
Max pooling		2×2	$56 \times 48 \times 64$	
Konvoluční 3.	128	3×3	$56 \times 48 \times 128$	73 856
Konvoluční 4.	128	3×3	$56 \times 48 \times 128$	147 584
Max pooling		2×2	$28 \times 24 \times 128$	
Konvoluční 5.	256	3×3	$28 \times 24 \times 256$	295 168
Konvoluční 6.	256	3×3	$28 \times 24 \times 256$	590 080
Konvoluční 7.	256	3×3	$28 \times 24 \times 256$	590 080
Max pooling		2×2	$14 \times 12 \times 256$	
Konvoluční 8.	512	3×3	$14 \times 12 \times 512$	1 180 160
Konvoluční 9.	512	3×3	$14 \times 12 \times 512$	2 359 808
Konvoluční 10.	512	3×3	$14 \times 12 \times 512$	2 359 808
Max pooling		2×2	$7 \times 6 \times 512$	
Konvoluční 11.	512	3×3	$7 \times 6 \times 512$	2 359 808
Konvoluční 12.	512	3×3	$7 \times 6 \times 512$	2 359 808
Konvoluční 13.	512	3×3	$7 \times 6 \times 512$	2 359 808
Max pooling		2×2	$3 \times 3 \times 512$	
Plně propojená 1.			4096	18 878 464
Dropout (0,5)			4096	
Plně propojená 2.			512	2 097 664
Dropout (0,5)			512	
Výstupní			10559	5 416 767

Tabulka 22: VGG16 architektura. U všech konvolučních vrstev je krok 1 a u max pooling je krok 2

B VGG19 architektura

Vrstva	**Počet filtrů**	**Velikost filtrů**	**Velikost výstupu**	**Počet parametrů**
Vstupní			$112 \times 96 \times 3$	
Konvoluční 1.	64	3×3	$112 \times 96 \times 64$	1792
Konvoluční 2.	64	3×3	$112 \times 96 \times 64$	36 928
Max pooling		2×2	$56 \times 48 \times 64$	
Konvoluční 3.	128	3×3	$56 \times 48 \times 128$	73 856
Konvoluční 4.	128	3×3	$56 \times 48 \times 128$	147 584
Max pooling		2×2	$28 \times 24 \times 128$	
Konvoluční 5.	256	3×3	$28 \times 24 \times 256$	295 168
Konvoluční 6.	256	3×3	$28 \times 24 \times 256$	590 080
Konvoluční 7.	256	3×3	$28 \times 24 \times 256$	590 080
Konvoluční 8.	256	3×3	$28 \times 24 \times 256$	590 080
Max pooling		2×2	$14 \times 12 \times 256$	
Konvoluční 9.	512	3×3	$14 \times 12 \times 512$	1 180 160
Konvoluční 10.	512	3×3	$14 \times 12 \times 512$	2 359 808
Konvoluční 11.	512	3×3	$14 \times 12 \times 512$	2 359 808
Konvoluční 12.	512	3×3	$14 \times 12 \times 512$	2 359 808
Max pooling		2×2	$7 \times 6 \times 512$	
Konvoluční 13.	512	3×3	$7 \times 6 \times 512$	2 359 808
Konvoluční 14.	512	3×3	$7 \times 6 \times 512$	2 359 808
Konvoluční 15.	512	3×3	$7 \times 6 \times 512$	2 359 808
Konvoluční 16.	512	3×3	$7 \times 6 \times 512$	2 359 808
Max pooling		2×2	$3 \times 3 \times 512$	
Plně propojená 1.			4096	18 878 464
Dropout (0,5)			4096	
Plně propojená 2.			512	2 097 664
Dropout (0,5)			512	
Výstupní			10559	5 416 767

Tabulka 23: VGG19 architektura. U všech konvolučních vrstev je krok 1 a u max pooling je krok 2

C Inception pomocný klasifikátor

Vrstva	Počet filtrů	Velikost filtrů	Velikost výstupu	Padding
Average pooling		5×5		Ano
Konvoluční	128	1×1	$4 \times 3 \times 128$	
Plně propojená			512	
Dropout (0,7)			512	
Výstupní			10559	

Tabulka 24: Pomocný klasifikátor, který je napojený na moduly 4a a 4d v Inception architektuře. Velikost výstupu první vrstvy záleží na vstupním modulu

D ResNet architektury

	ResNet34	ResNet50	Velikost výstupu
	Konvoluční 7×7, 64 filtrů, krok 2		56×48
	Max pooling 3×3, krok 2		27×23
Sekce 1	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	27×23
Sekce 2	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	14×12
Sekce 3	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	7×6
Sekce 4	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	4×3
	Average pooling 4×3		1×1
	Plně propojená		10559

Tabulka 25: ResNet34 architektura. Reziduální bloky (obrázek 13) jsou označeny hranatými závorkami. Zmenšení příznakových map se provádí v sekcích 2, 3 a 4 v první konvoluční vrstvě s krokem 2. Batch normalizace je aplikována po každé konvoluční vrstvě (před aktivační funkcí)

E Obsah odevzdané přílohy

- **dokumentace - latex** - elektronická verze dokumentace ve formátu LaTeX a PDF
- **datasety** - předzpracovaný datasest LFW-aligned, ukázka datasetu CASIA-aligned a odkaz na Google Drive pro stažení ostatních datasetů
- **váhy** - váhy nejlepší natrénované sítě
- **implementace** - python skripta pro spuštění trénování, testování, vytvoření grafů a predzpracovani dat
- **help.txt** - postup pro nainstalování knihoven a puštění skript